Inductive Temporal Logic Programming

Dipl. Inf. Robert Kolter

Dem Fachbereich Informatik
der Technischen Universität Kaiserslautern
vorgelegte Dissertation
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften

Kaiserslautern, 6.12.2009

Dekan:	Prof. Dr. Karsten Berns
Berichterstatter:	Prof. Dr. Otto Mayer
	Prof. Dr. Markus Nebel

Tag der wissenschaftlichen Aussprache: 29.5.2009

D386

Contents

1. Introduction **1**

I. Temporal Logic and (inductive) Logic Programming **5**

2. Preliminaries **7**

2.1. Propositional Logic 7

2.2. First Order Logic 12

2.3. Temporal Logic 19

2.3.1. Propositional Temporal Logic 20

2.3.2. First Order Temporal Logic 23

3. Logic Programming **29**

3.1. Predicate Logic as a Programming Language 29

3.2. The Concept of SLD–Resolution 33

3.3. Soundness and Completeness 40

4. Inductive Logic Programming **43**

4.1. The basic Framework 43

4.2. Generalization and Specialization 46

4.3. Refinement Operators and their Properties 47

II. First Order Inductive Temporal Logic Programming 49

5. The Programming Language Prolog(+T) 51

5.1. Syntax of PROLOG(+T) 52

5.1.1. Terms in PROLOG(+T) 52

5.1.2. Facts 55

5.1.3. Rules 55

5.1.4. Programs 57

5.1.5. Queries 59

5.1.6. The Relation of PROLOG(+T) to other Temporal Logic programming Languages 60

5.2. Declarative Semantics of PROLOG(+T) 65

5.2.1. Preliminaries 65

5.2.2. Reduction of Literals 74

5.2.3. Semantics for programs 81

5.3. A Saturation–based temporal Proof Procedure 86

5.3.1. Tableaux Rules 87

5.3.2. Soundness and Completeness Issues 104

6. The Lattice Structure of Prolog(+T) objects 115

6.1. Subsumption 116

6.2. The Lattice Structure of Literals 117

6.2.1. Generalizations of Terms 117

6.2.2. Generalizations and Specializations of Literals 119

6.3. The Lattice Structure of Rules 125

6.3.1. Greatest Specializations 125

6.3.2. Least Generalizations 129

7. Refinement Operators for Prolog(+T)–programs 141

7.1. Refinement Operators for PROLOG(+T)–Literals 142

7.2. Refinement Operators for Rules 147

7.2.1. Downward Refinement 150

7.2.2. Upward Refinement 152

7.3. Refinement Strategies 159

7.3.1. Elimination of Variants 162

7.3.2. Restriction to reduced Rules 163

7.3.3. Elimination of Tautologies 165

7.3.4. Premises vs. Conclusions 166

8. Identifiability of Prolog(+T)–programs 169

8.1. PAC–Learning 170

8.2. Learnability 177

8.2.1. The general case 177

8.2.2. Programs with syntactical restrictions 183

III. Propositional Inductive Temporal Logic Programming 189

9. Preliminaries 191

9.1. Finite Automata on infinite Objects 192

9.2. Automata Constructions for Propositional Temporal Logic Formulas . . . 196

9.2.1. A Modified Formal Automata–Model 196

9.2.2. A primitive Construction 199

9.2.3. An Overview over improved Constructions 203

9.2.4. Some Complexity Results 205

9.2.5. Checking Language–Emptiness 210

10. Automata Manipulations **211**

10.1. Implication as an Ordering 212

10.2. Upward Refinement 213

10.2.1. Formulas without Temporal Operators 214

10.2.2. Formulas with Temporal Operators 216

10.3. Downward Refinement 218

10.3.1. Formulas without Temporal Operators 219

10.3.2. Formulas with Temporal Operators 220

10.4. Modifying Automata by Application of Refinement Operations 221

10.4.1. Upward Refinement 221

10.4.2. Downward Refinement 232

10.4.3. Extracting Formulas 237

10.5. The Identification Process 238

11. Identifiability of Ltl–programs **251**

11.1. General Notations 251

11.2. Upper Bounds for the VC–Dimension of Classes of LTL–Programs 254

11.2.1. General LTL–Programs 254

11.2.2. The deterministic Sublogic of LTL 256

12. Conclusions **261**

12.1. Summary of the Topics 261

12.2. Perspectives for Future Research 263

A. Formal Description of the Programming Languages **267**

A.1. Syntax of PROLOG(+T) 267

A.1.1. Terms 268

A.1.2. Atoms and Literals 270

A.1.3. Rules 271

A.1.4. General PROLOG(+T)–Objects 271

A.2. Syntax of LTL 272

1. Introduction

The field of *Inductive Logic Programming* (see [121], [119], [120], [118], [99] and [126]) is an area of active research. Inductive Logic Programming (ILP) deals with the topic of extracting a suitable explanation of a phenomenon from finite sets of examples. While this is the basic topic in most areas of *Artificial Intelligence* (AI), the target concept in ILP (i.e. the phenomenon to be *learned*) is a logic program. Logic programs are finite sets of clauses which allow a very natural interpretation as declarations of procedural rules. While this natural interpretation of a clause

$$A \leftarrow B_1, \ldots, B_n$$

is "*to solve the task A, solve the sequence $B_1, \ldots, B_n$ of subtasks*", the interpretation in ILP is more *rule* based: if $x_1, \ldots, x_n$ are the variable symbols occurring in the *head* of the clause (i.e. in A), then $x_1, \ldots, x_n$ will be assumed to have the *property* A if they have the properties $B_1, \ldots, B_n$. This gives the interpretation "*if $B_1, \ldots, B_n$ hold, then A does also hold*".

Since the concepts of interest are logic programs, examples for any ILP–based learning system can (or should) be *ground atoms*. Each such example can be either a *positive* or a *negative* example: if P is the program to be learned and e is an example, then

- if e is positive, then $P \models e$ and
- if e is negative, then $P \not\models e$.

ILP techniques have been applied in many branches of science, notable in computational biology. There, several areas of application have been identified, notably

- Drug–activity comparison (see [88], [148], [24] and [87]),
- Diagnosis of rheumatic diseases (see [33] and [100]),
- Mutagenesis prediction (see [40], [151], [152] and [153]),
- Protein–structure prediction (see [122] and [86]) and
- Design of Medical Diagnosis Tools (see [136] and [101]).

Other areas of application are prediction of strategies for chess games (see [16]), finite element methods (see [50], [52], [51], [54] and [55]), data mining (see [173], [115] and [116]) and learning models for dynamic systems (see [25] and [55]).

This thesis deals with a natural extension of the classical ILP paradigm, namely synthesizing *temporal* logic programs from given examples. Temporal Logic is a natural way to describe relations which may change over time. We will distinguish two different areas:

Propositional Inductive Temporal Logic Programming Here the language used in order to describe the programs is LTL, a simple temporal logic language allowing the use of operators such as X, G, F, U and R for modeling time–dependent relations. LTL is very popular in the fields of *Model Checking* and *Supervisory Control*. We will exploit the fact that each LTL–formula can be represented as a nondeterministic Büchi–automaton (an automaton accepting infinite sequences of letters) in order to define operators which manipulate such an automaton in order to fit the specification given by the examples.

First Order Inductive Temporal Logic Programming Here the language of interest is much more flexible and expressive than in the propositional case. The major drawback is the undecidability of first order logic (which has been proven by Church in

1936) and therefore of the full first order temporal logic. To keep first order temporal logic tractable, we identify a simple extension of a Prolog–style language which allows the usage of temporal the operators X, G, F, U and P in front of literals. We will see that this programming language (which we will call Prolog(+T)) has a very natural procedural interpretation in terms of some well known rewrite rules for temporal logic formulas. These rules will be used in order to give a *saturation* based calculus.

The main part of this thesis will be devoted to the field of First Order Inductive Temporal Logic Programming. We will see that the lattice properties (with respect to the *subsumption*–ordering) of first order atoms, literals and clauses can be extended to Prolog(+T)–atoms, –literals and –clauses. So the existence of least generalizations and greatest specializations of Prolog(+T)–clauses can be ensured. We will give algorithms which allow the computation of such generalizations and specializations and use the techniques from these algorithms in order to define *refinement operators* for Prolog(+T) programs.

The thesis is structured as follows: in the first part we will define some basic notations from first order logic, temporal logic and logic programming. The chapters in this first part are kept rather short since we assume that the reader is familiar with these topics. After having introduced these basic concepts we will briefly introduce some concepts from the field of ILP.

The second part is dedicated to an in–depth treatment of First Order Temporal ILP. This includes the definition of Prolog(+T) and the discussion of its declarative semantics. Having achieved this, we will present a proof procedure, discuss the lattice properties of Prolog(+T) objects and study refinement operators.

The third part is then dedicated to Propositional Temporal ILP. After having defined basic concepts from the field of ω–automata, we will present two operations for refining Ltl–programs by manipulating their representing automata. The final chapter of this third part will be devoted to the question of the complexity of the identification task. Therefore we will derive upper bounds for the VC–dimension of certain classes of Ltl–programs. These VC–dimensions allow a direct extraction of the number of examples which are needed in order to identify the program under consideration (or more precise: a program which is equivalent to the program under consideration).

Part I.

Temporal Logic and (inductive) Logic Programming

2. Preliminaries

Contents

2.1. Propositional Logic **7**

2.2. First Order Logic **12**

2.3. Temporal Logic **19**

2.3.1. Propositional Temporal Logic 20

2.3.2. First Order Temporal Logic 23

In this chapter we will briefly define and review some of the basic and most important concepts which we will use throughout the rest of this thesis. This includes propositional logic, first order logic and temporal logic. Each of these three logics is equipped with both a semantical consequence relation which we will as usual denote as $\models$. The three logics will be defined by first defining the sets of formulas which can be built from some atomic objects and connectives and can be seen as a more or less detailed way to describe mathematical concepts in a syntactic way. The properties of the logics are only mentioned. We will not prove them since the literature on propositional, first order and temporal logic is rich (see [18], [147], [31], [20], [89] and [56]).

2.1. Propositional Logic

The simplest logic which we will define is the classical *propositional logic*. Propositional logic is a formalism which has been studied very well. Early studies were done by

Boole (see [23]) in the 19th century. But to this date the study of propositional logic was motivated by the circumstance that researchers wanted to formalize the process of mathematical reasoning. Propositional logic seemed to be a good starting point for such formalizations. Later the usefulness of propositional logic for the description of electrical and electronical circuits was pointed out. We refer to [104] for an introduction to applications of propositional logic.

Now assume that a countable infinite set $X = \{p_i \mid i \in \mathbb{N}\}$ is given. Each element of X will be called a ***propositional variable*** or simply a ***variable*** if there is no way of confusion. *Formulas* of the propositional language defined over X are defined inductively as follows (roughly following the treatment from [20]).

Definition 2.1.1 (Propositional Logic)

Let a set X of propositional variables be given. The set of *formulas* over X is defined as:

1. `true` and `false` are formulas,
2. each $p \in X$ is a formula,
3. if φ is a formula, then $\neg\varphi$ is a formula and
4. if φ_1 and φ_2 are formulas, then so are $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$.

The set of *all* formulas over X will be denoted as $\mathcal{F}(X)$.

The above definition of propositional formulas models the syntactic level of propositional reasoning. To model the semantic part, i.e. the logical consequence relation, we will now define a way to *evaluate* propositional formulas to values 1 (identified as *true*) and 0 (identified as *false*). This will be done using a suitable concept of *evaluation* *functions.*

Definition 2.1.2 (Evaluation)

Let X be a set of propositional variables. An *evaluation* for X is a function $v : X \cup \{\texttt{true}, \texttt{false}\} \to \mathbb{Z}_2$ (where $\mathbb{Z}_2$ denotes the Galois field of cardinality 2) satisfying $v(\texttt{true}) = 1$ and $v(\texttt{false}) = 0$.

The concept of an evaluation can now be extended to functions $\hat{v} : \mathcal{F}(X) \to \mathbb{Z}_2$ in the obvious (homomorphic) way. Let $\varphi \in \mathcal{F}(X)$ be any formula. Then

1. if $\varphi = p \in X$, then $\hat{v}(p) = v(p)$,
2. if $\varphi = \neg\psi$, then $\hat{v}(\varphi) = 1 - \hat{v}(\varphi)$,
3. if $\varphi = (\varphi_1 \wedge \varphi_2)$, then $\hat{v}(\varphi) = \min\{\hat{v}(\varphi_1), \hat{v}(\varphi_2)\}$,
4. if $\varphi = (\varphi_1 \vee \varphi_2)$, then $\hat{v}(\varphi) = \max\{\hat{v}(\varphi_1), \hat{v}(\varphi_2)\}$,
5. if $\varphi = (\varphi_1 \to \varphi_2)$, then $\hat{v}(\varphi) = \max\{\hat{v}(\neg\varphi_1), \hat{v}(\varphi_2)\}$ and
6. if $\varphi = (\varphi_1 \leftrightarrow \varphi_2)$, then $\hat{v}(\varphi) = \min\{\hat{v}(\varphi_1 \to \varphi_2), \hat{v}(\varphi_2 \to \varphi_1)\}$.

Since there is no way of confusion we will from now on identify $\hat{v}$ and v writing $v(\varphi)$ for the result of $\hat{v}(\varphi)$ for any formula $\varphi \in \mathcal{F}(X)$. The set of all evaluations (or *valuations* from now on) will be denoted as $\textsc{Val}$.

For the sake of simplicity we will introduce two more concepts:

1. Let $\varphi_1, \ldots, \varphi_n$ be any finite sequence of formulas from $\mathcal{F}(X)$. Then the formulas $\bigwedge_{i=1}^{n} \varphi_i$ and $\bigvee_{i=1}^{n} \varphi_i$ are defined as

$$\begin{aligned}
\bigwedge_{i=1}^{n} \varphi_i &= \left(\varphi_n \wedge \bigwedge_{i=1}^{n-1} \varphi_i\right) \text{ and} \\
\bigvee_{i=1}^{n} \varphi_i &= \left(\varphi_n \vee \bigvee_{i=1}^{n-1} \varphi_i\right).
\end{aligned}$$

2. We assume that the connectives have the following *binding priorities*:

   a) $\neg$ has a higher binding priority than $\wedge$,

   b) $\wedge$ has a higher binding priority than $\vee$,

   c) $\vee$ has a higher binding priority than $\rightarrow$ and

   d) $\rightarrow$ has a higher binding priority than $\leftrightarrow$.

   Applying the binding priority rules allows the omission or brackets in many formulas which improves the readability. For example, the formula

   $$(\neg p_1 \vee (p_2 \wedge p_3)) \rightarrow p_4$$

   can be written as

   $$\neg p_1 \vee p_2 \wedge p_3 \rightarrow p_4$$

Now let φ be any formula. A valuation v is called a *model* of φ if and only if $v(\varphi) = 1$. The set of all models of φ will be denoted as $MD(\varphi)$[1]. Similarly for sets Φ of formulas we define a valuation v to be a model of Φ if $v(\varphi) = 1$ for every $\varphi \in \Phi$. We then have $MD(\Phi) = \bigcap_{\varphi\in\Phi} MD(\varphi)$.

We will call pairs $(\varphi_1, \varphi_2) \in \mathcal{F}(X)^2$ (semantically) *equivalent* (written $\varphi_1 \equiv \varphi_2$) if and only if $v(\varphi_1) = v(\varphi_2)$ for every valuation v. Equivalently we could define semantical equivalence as follows:

$$\begin{aligned} \varphi_1 \equiv \varphi_2 \quad & \text{if and only if} \quad \{\varphi_1\} \models \varphi_2 \text{ and } \{\varphi_2\} \models \varphi_1 \\ & \text{if and only if} \quad \emptyset \models \varphi_1 \leftrightarrow \varphi_2 \end{aligned}$$

[1]So $MD(\varphi) = \{v \in \textsc{Val} \mid v(\varphi) = 1\}$.

where $\models$ denotes the semantical consequence relation which will be defined below.

Note that $\equiv$ is an equivalence relation on $\mathcal{F}(X)$. Furthermore note that $\varphi_1 \equiv \varphi_2$ if and only if $MD(\varphi_1) = MD(\varphi_2)$.

A formula $\varphi \in \mathcal{F}(X)$ is called

- *satisfiable* if $MD(\varphi) \neq \emptyset$,
- *valid* if $MD(\varphi) = \text{VAL}$ and
- *unsatisfiable* if $MD(\varphi) = \emptyset$.

Similar concepts can be defined for sets of formulas. A set $\Phi \subseteq \mathcal{F}(X)$ is called

- *satisfiable* if $MD(\Phi) \neq \emptyset$,
- *valid* if $MD(\Phi) = \text{VAL}$ and
- *unsatisfiable* if $MD(\Phi) = \emptyset$.

Satisfiable sets of propositional formulas can be characterized by Theorem 2.1.1 which is commonly known as the *finiteness theorem.*

Theorem 2.1.1 (Finiteness Theorem)
Let $\Phi \subseteq \mathcal{F}(X)$ be a set of formulas. Then Φ is satisfiable if and only if every finite set $\Psi \subseteq \Phi$ is satisfiable.

The concept of *logical consequence* is commonly modeled as follows: A formula φ is a logical consequence of a set $\Phi \subseteq \mathcal{F}(X)$ if every model of Φ is also a model of φ.

Definition 2.1.3 (Logical Consequence)
Let $\Phi \subseteq \mathcal{F}(X)$ be a set of formulas and let $\varphi \in \mathcal{F}(X)$ be a formula. Then φ is a logical consequence of Φ (written as $\Phi \models \varphi$) if and only if for every $v \in \text{VAL}$ such as $v(\Phi) \subseteq \{1\}^2$ it holds that $v(\varphi) = 1$.

Equivalently one can define $\Phi \models \varphi$ if and only if $MD(\Phi) \subseteq MD(\varphi)$. Moreover we can define the following: Let Φ be as above and let $\Psi \subseteq \mathcal{F}(X)$ be any set of formulas. Then $\Phi \models \Psi$ if and only if $\Phi \models \psi$ for every $\psi \in \Psi$. If Ψ is a finite set, say $\Psi = \{\psi_1, \ldots, \psi_n\}$, then $\Phi \models \Psi$ if and only if $\Phi \models \bigwedge_{i=1}^{n} \psi_i$.

Equivalently one can use the following characterization of unsatisfiable sets of formulas.

Theorem 2.1.2

Let $\Phi \subseteq \mathcal{F}(X)$ be a set of formulas. Then Φ is unsatisfiable if and only if $\Phi \models \texttt{false}$.

The logical consequence relation enjoys the following nice properties.

Theorem 2.1.3

Let $\Phi \subseteq \mathcal{F}(X)$ be a set of formulas and let $\varphi \in \mathcal{F}(X)$ be a formula. Then

1. $\Phi \models \varphi$ if and only if $\Phi \cup \{\neg\varphi\} \models \texttt{false}$.

2. If $\Phi \models \texttt{false}$, then there is a finite set $\Psi \subseteq \Phi$ such that $\Psi \models \texttt{false}$.

Part 2 of Theorem 2.1.3 is also known as the *compactness theorem*. It is an easy corollary of Theorem 2.1.1.

For finite sets $\Phi \subseteq \mathcal{F}(X)$ of propositional logic formulas and formulas $\varphi \in \mathcal{F}(X)$ it is *decidable* whether $\Phi \models \varphi$ holds or not. However, the related *satisfiability* problem is *$\mathcal{NP}$–complete* (see [38] and [71]), that is one cannot (or better *should not*) hope for efficient procedures which are capable of deciding if a formula φ (or a set of Φ formulas) is satisfiable. Moreover the problem of deciding the logical consequence relation is *$co - \mathcal{NP}$–complete* which indicates that is in some sense even *more difficult* to decide than satisfiability.

2.2. First Order Logic

In contrast to propositional logic, *first order logic* or *(first order) predicate logic* allows a more precise formalization of (mathematical) relations. Using a more flexible language of logic was motivated by the limitations arising from the usage of propositional logic. For example propositional logic only allows reasoning about propositions, i.e. things which are either *true* or *false*. But in Mathematics the truth or falsity of a proposition often depends on the values of certain variables occurring as inputs to functions. For example the formula $f(x) = 0$ should be evaluable to true or false. But the truth–value of this formula depends on

- the function which is represented by the function–symbol f and
- the value which is assigned to the variable x.

So propositional logic is not an adequate formal system for modeling this formula. Early studies of first order logic were presented at the beginning of the 20th century e.g. by Frege (see [69]), Gentzen (see [75]), Russell and Whitehead (see [170]) and several others. At this point of time sound and complete calculi have been developed. But practical applications arose much later.

As we have already pointed out formulas are not built from *propositions* alone but from a more general concept which we will call *atomic formulas* or simply *atoms* from now on. Therefore we will have to refine the syntax of the logical language to be used in a suitable way.

Recall that in the case of propositional logic, the syntax (i.e. the formulas of a logical language) only depends on the set X of propositional variables. A similar concept for first order logic is given by the concept of *signatures*.

Definition 2.2.1 (Signature)

A signature is a tuple $\text{sig} = (\mathcal{X}, F, P, \alpha)$ where

1. $\mathcal{X} = \{x_i \mid i \in I \text{ for some set } I \subseteq \mathbb{N} \text{ of indices}\}$ is a countable set of *variable symbols*,
2. F and P are finite sets of *function*– resp. *predicate–symbols* and

3. $\alpha : F \cup P \to \mathbb{N}$ is a function which maps each symbol to a natural number (its *arity*) and is therefore called the *arity–function*.

If $\sigma \in F \cup P$ is a function– respectively predicate–symbol and $\alpha(\sigma) = n$ for some $n \geq 0$, we will say that σ *has arity* n. In the case that $n = 0$ we will also say that σ is a *constant symbol*.

The simplest objects which can be built from a signature are *terms*.

Definition 2.2.2 (Terms)
Let $\mathrm{sig} = (\mathcal{X}, F, P, \alpha)$ be a signature. The set $\mathcal{T}(\mathrm{sig})$ (or simply $\mathcal{T}$) of *terms* over sig is defined inductively as follows:

1. each $x \in \mathcal{X}$ is a term and
2. if $f \in F$ is a function–symbol, $n = \alpha(f)$ and $t_1, \ldots, t_n \in \mathcal{T}$ are terms, then so is $f(t_1, \ldots, t_n)$.

Formulas are now defined to be either *atomic* ones or formulas composed from simpler ones.

Definition 2.2.3 (Atomic Formulas)
Let $\mathrm{sig} = (\mathcal{X}, F, P, \alpha)$ be a signature. The set $\mathcal{A}(\mathrm{sig})$ (or simply $\mathcal{A}$) of all *atomic formulas* (or simply *atoms*) over sig is defined inductively as follows:

1. `true` and `false` are in $\mathcal{A}$ and
2. if $p \in P$ is a predicate symbol, $n = \alpha(p)$ and $t_1, \ldots, t_n \in \mathcal{T}$ are terms, then $p(t_1, \ldots, t_n)$ is in $\mathcal{A}$.

In contrast to formulas of the propositional logic language, first order logic formulas are also capable to model terms like "for all x it holds that ..." and "there is an x such that …". This is achieved by introducing two *quantifiers* $\forall$ (the *universal quantifier*) and $\exists$ (the *existence quantifier*).

Definition 2.2.4 (First Order Formulas)

Let sig $= (\mathcal{X}, F, P, \alpha)$ be a signature. The set $\mathcal{F}(\text{sig})$ (or simply $\mathcal{F}$) of *formulas* over sig is defined inductively as follows:

1. every $\varphi \in \mathcal{A}$ is a formula,
2. if φ is a formula, then so is $\neg\varphi$,
3. if φ_1 and φ_2 are formulas, then so are $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$ and
4. if φ is a formula and $x \in \mathcal{X}$ is a variable symbol, then $\forall x\varphi$ and $\exists x\varphi$ are formulas.

Formulas from the set

$$\mathcal{A} \cup \{\neg\varphi \mid \varphi \in \mathcal{A}\}$$

are called *literals*. Since the formulas of first order logic allow finer reasoning about mathematical concepts, their *interpretation* also has to be more detailed. This includes the interpretation of the function symbols and the interpretation of the predicate symbols. So the concept of *evaluation functions* as introduced for propositional logic is not adequate anymore. An interpretation is given as a tuple consisting of a set of possible *values* of the variables (the *universe*), two mappings assigning functions to the function symbols and predicates to the predicate symbols and a mapping assigning values to the variable symbols.

Definition 2.2.5 (Interpretation)

Let sig $= (\mathcal{X}, F, P, \alpha)$ be a signature. An *interpretation* (for sig–formulas) is a tuple

$$\mathcal{J} = (U_{\mathcal{J}}, \mathfrak{F}, \mathfrak{P}, w)\,,$$

where

1. $U_{\mathcal{J}}$ is a nonempty set of objects (the *universe* of $\mathcal{J}$),
2. $\mathfrak{F}$ is a function which maps each $f \in F$ to a function $f^{\mathcal{J}} : U_{\mathcal{J}}^{\alpha(f)} \to U_{\mathcal{J}}$,
3. $\mathfrak{P}$ is a function which maps each $p \in F$ to a predicate $p^{\mathcal{J}} : U_{\mathcal{J}}^{\alpha(p)} \to \{0,1\}$ and
4. $w : \mathcal{X} \to U_{\mathcal{J}}$ maps each variable symbol x to an element $w(x) \in U_{\mathcal{J}}$.

Now let $\mathcal{J}$ be an interpretation. The *evaluation of terms* is straightforward depending on the structure of the term to be evaluated:

1. $\mathcal{J}(x) =: x^{\mathcal{J}} = w(x)$ for every $x \in \mathcal{X}$ and
2. $\mathcal{J}(f(t_1, \ldots, t_n)) = f^{\mathcal{J}}\left(t_1^{\mathcal{J}}, \ldots, t_n^{\mathcal{J}}\right)$ for each $f(t_1, \ldots, t_n) \in \mathcal{T}$.

Having defined how to evaluate terms, every formula can be evaluated in a straightforward way:

1. $\mathcal{J}\left(p(t_1, \ldots, t_n)\right) = p^{\mathcal{J}}\left(t_1^{\mathcal{J}}, \ldots, t_n^{\mathcal{J}}\right) \in \{0,1\}$,
2. $\mathcal{J}(\neg\varphi) = 1 - \mathcal{J}(\varphi)$,
3. $\mathcal{J}(\varphi_1 \wedge \varphi_2) = \min\{\mathcal{J}(\varphi_1), \mathcal{J}(\varphi_2)\}$,
4. $\mathcal{J}(\varphi_1 \vee \varphi_2) = \max\{\mathcal{J}(\varphi_1), \mathcal{J}(\varphi_2)\}$,
5. $\mathcal{J}(\varphi_1 \to \varphi_2) = \max\{\mathcal{J}(\neg\varphi_1), \mathcal{J}(\varphi_2)\}$,

6. $\mathcal{J}(\varphi_1 \leftrightarrow \varphi_2) = \min\{\mathcal{J}(\varphi_1 \rightarrow \varphi_2), \mathcal{J}(\varphi_2 \rightarrow \varphi_1)\}$,
7. $\mathcal{J}(\forall x \varphi) = \min\left\{\mathcal{J}_x^t(\varphi) \mid t \in U_{\mathcal{J}}\right\}$ and
8. $\mathcal{J}(\exists x \varphi) = \max\left\{\mathcal{J}_x^t(\varphi) \mid t \in U_{\mathcal{J}}\right\}$.

where $\mathcal{J}_x^t$ is defined as $\mathcal{J}_x^t = (U_{\mathcal{J}}, \mathfrak{F}, \mathfrak{P}, \tilde{w})$ with $\tilde{w} : \mathcal{X} \rightarrow U_{\mathcal{J}}$ being defined as

$$\tilde{w}(\bar{x}) = \begin{cases} w(\bar{x}) & \Leftrightarrow \bar{x} \neq x \\ t & \Leftrightarrow \bar{x} = x \end{cases}.$$

We will write $\mathcal{J} \models \varphi$ if $\mathcal{J}(\varphi) = 1$. Similarly we will write $\mathcal{J} \models \Phi$ (for a set $\Phi \subseteq \mathcal{F}$) if and only if $\mathcal{J} \models \varphi$ for every $\varphi \in \Phi$.

As in the case of propositional logic an interpretation $\mathcal{J}$ is called a *model* of a formula φ (respectively a *model* of a set Φ of formulas) if $\mathcal{J} \models \varphi$ (respectively $\mathcal{J} \models \Phi$). The set of all models of φ is defined to be

$$MD(\varphi) = \{\mathcal{J} \mid \mathcal{J} \models \varphi\}$$

and the set of all models for Φ is

$$MD(\Phi) = \bigcap_{\varphi \in \Phi} MD(\varphi).$$

We will adopt the following terms from propositional logic: A formula $\varphi \in \mathcal{F}$ is called

- *satisfiable* if $MD(\varphi) \neq \emptyset$,
- *valid* if $\mathcal{J} \models \varphi$ for every interpretation $\mathcal{J}$ and
- *unsatisfiable* if $MD(\varphi) = \emptyset$.

Similar concepts can be defined for sets of formulas. A set $\Phi \subseteq \mathcal{F}$ is called

- *satisfiable* if $MD(\Phi) \neq \emptyset$,
- *valid* if every $\varphi \in \Phi$ is valid and
- *unsatisfiable* if $MD(\Phi) = \emptyset$.

As in the case of propositional logic we define a relation $\models$ by defining:

Definition 2.2.6 (Logical Consequence)

Let $\Phi \subseteq \mathcal{F}$ be a set of formulas and let $\varphi \in \mathcal{F}$ be a formula. Then φ is a logical consequence of Φ ($\Phi \models \varphi$) if and only if for every interpretation $\mathcal{J}$ such as $\mathcal{J}(\Phi) = 1$ it holds that $\mathcal{J}(\varphi) = 1$.

Again we can express $\models$ by $\Phi \models \varphi$ if and only if $MD(\Phi) \subseteq MD(\varphi)$.

The properties of $\models$ are still the same as in propositional logic.

Theorem 2.2.1

Let $\Phi \subseteq \mathcal{F}$ be a set of formulas. Then Φ is unsatisfiable if and only if $\Phi \models$ `false`.

Theorem 2.2.2

Let $\Phi \subseteq \mathcal{F}$ be a set of formulas. Then Φ is satisfiable if and only if every finite set $\Phi' \subseteq \Phi$ is satisfiable.

Theorem 2.2.3

Let $\Phi \subseteq \mathcal{F}$ be a set of formulas and let $\varphi \in \mathcal{F}$ be a formula. Then

1. $\Phi \models \varphi$ if and only if $\Phi \cup \{\neg\varphi\} \models$ `false`.
2. If $\Phi \models$ `false`, then there is a finite set $\Psi \subseteq \Phi$ such that $\Psi \models$ `false`.

In contrast to propositional logic, where testing for satisfiability is decidable (but $\mathcal{NP}$–complete), testing for satisfiability is undecidable in first order logic. This is due to the following theorem proved by Church in 1936 (see [32]).

Theorem 2.2.4

Let $\Phi \subseteq \mathcal{F}$ be a set of formulas. Then the following problem is undecidable:

Input: Φ

Output: $\begin{cases} 1 & \Leftrightarrow MD(\Phi) \neq \emptyset \\ 0 & \Leftrightarrow \text{else} \end{cases}$

2.3. Temporal Logic

In contrast to the logics which we have introduced so far, *temporal logic* is concerned with reasoning about *time–dependent properties.* An example might be the operator X which has the following intuitive interpretation:

If φ is true at the next point of time, then $\mathsf{X}\varphi$ is true at the current point of time.

Consequently X will be referred to as *Next–State–Operator.* Here we can already notice that in our temporal logics time will be of discrete nature. Consequently any sequence of *points of time* can only contain countably many such *points.*

Again we wish to distinguish temporal logics according to the primitive objects under consideration. So we will have *propositional temporal logic* and *first order temporal logic* as alternatives while the former is properly contained in the latter. Another possible criterion of differentiation is between *linear time* (see [110]) and *branching time* (see [58], [59]) logics. Linear time logics allow reasoning about *one* possible continuation of the current point of time while branching time logics are equipped with operators quantifying over sequences of continuations, so called *paths* (and are therefore called *path quantifiers*). We will only consider linear time temporal logics since they are well suited for our purposes. Perhaps the most prominent of these linear time temporal logics is LTL which has been subject of both theoretical research and practical applications.

Since temporal logics allow reasoning about time–dependent aspects of objects, the concept of interpretations will have to be extended to *sequences of interpretations.* This will be the subject of consideration for the rest of this chapter.

2.3.1. Propositional Temporal Logic

Again assume that X is a given set of *propositional variables* as defined in section 2.1, that is $X = \{p_i \mid i \in \mathbb{N}\}$. The language LTL of *linear time temporal logic formulas* is built from the language $\mathcal{F}(X)$ by introducing several *temporal operators*.

Definition 2.3.1 (Propositional Linear Time Temporal Logic, e.g. [110]) The language LTL of *linear time temporal logic formulas* is inductively defined as

1. every $\varphi \in \mathcal{F}(X)$ is in LTL,
2. if φ is in LTL, then so are $\mathsf{X}\varphi$, $\mathsf{G}\varphi$ and $\mathsf{F}\varphi$ and
3. if φ_1, φ_2 are in LTL, then so are $\varphi_1 \mathsf{U} \varphi_2$ and $\varphi_1 \mathsf{R} \varphi_2$

The temporal operators X, G, F, U and R will have the following intuitive interpretation:

1. $\mathsf{X}\varphi$: if φ is true at the next point of time, then $\mathsf{X}\varphi$ is true at the actual point of time (*Next–State–Operator*).
2. $\mathsf{G}\varphi$: φ is true at *every* point of time (*Always–Operator*).
3. $\mathsf{F}\varphi$: there is a point of time such that φ is true at this point (*Eventually–Operator*).
4. $\varphi_1 \mathsf{U} \varphi_2$: φ_1 holds *until* φ_2 is true (*Until–Operator*).
5. $\varphi_1 \mathsf{R} \varphi_2$: φ_1 has to be true *before* φ_2 is true (*Release–Operator*).

Formally LTL–formulas are evaluated in *sequences of states* each of which is a single evaluation of the propositional symbols of the language which is defined by the set X. Sequences of states are assumed to be

- infinite and

- countable.

So each such sequence is isomorphic to the set $\mathbb{N}$ of natural numbers and the time points in these sequences are *discrete*. Consequently the set of *all* sequences of states is uncountable.

Definition 2.3.2 (Temporal State)
A *temporal state* is a set $s \subseteq X$.

We can interpret a temporal state s as an evaluation $v_s : X \to \mathbb{Z}_2$ defined by

$$v_s(x) = \begin{cases} 1 & \Leftrightarrow x \in s \\ 0 & \Leftrightarrow \text{else} \end{cases}$$

and extend this evaluation from X to $\mathcal{F}(X)$ in the obvious way.

So far we are not able to assign a meaning to the temporal operators G, F, X, U and P. Therefore we extend the concept of an *evaluation* (as introduced in section 2.1) to *temporal interpretations* defined formally as follows.

Definition 2.3.3 (Temporal Interpretation)
A *temporal interpretation* (or *interpretation* for short) is an infinite sequence $\mathcal{J} = (s_0, s_1, \ldots, s_i, \ldots)$ of temporal states.

For $j \in \mathbb{N}$ the notation $\mathcal{J}^j$ will denote the temporal interpretation starting at time point j, i.e. $\mathcal{J}^j = (s_j, s_{j+1}, \ldots, s_k, \ldots)$.

Now let $\varphi \in$ LTL be a formula and let $\mathcal{J} = (s_0, s_1, \ldots, s_i, \ldots)$ be a temporal interpretation. We extend the relation $\models$ as follows:

1. if $\varphi \in X$, then $\mathcal{J} \models \varphi$ if and only if $\varphi \in s_0$,

2. if $\varphi = \neg\psi$, then $\mathcal{J} \models \varphi$ if and only if $\mathcal{J} \not\models \psi$,
3. if $\varphi = \varphi_1 \wedge \varphi_2$, then $\mathcal{J} \models \varphi$ if and only if $\mathcal{J} \models \varphi_1$ and $\mathcal{J} \models \varphi_2$,
4. if $\varphi = \varphi_1 \vee \varphi_2$, then $\mathcal{J} \models \varphi$ if and only if $\mathcal{J} \models \varphi_1$ or $\mathcal{J} \models \varphi_2$,
5. if $\varphi = \varphi_1 \rightarrow \varphi_2$, then $\mathcal{J} \models \varphi$ if and only if $\mathcal{J} \not\models \varphi_1$ or $\mathcal{J} \models \varphi_2$,
6. if $\varphi = \mathsf{X}\psi$, then $\mathcal{J} \models \varphi$ if and only if $\mathcal{J}^1 \models \psi$,
7. if $\varphi = \mathsf{G}\psi$, then $\mathcal{J} \models \varphi$ if and only for every $i \geq 0$ it holds that $\mathcal{J}^i \models \psi$,
8. if $\varphi = \mathsf{F}\psi$, then $\mathcal{J} \models \varphi$ if and only if there is $i \geq 0$ such that $\mathcal{J}^i \models \psi$,
9. if $\varphi = \varphi_1\mathsf{U}\varphi_2$, then $\mathcal{J} \models \varphi$ if and only if there is $i \geq 0$ such that $\mathcal{J}^i \models \varphi_2$ and for every j such that $0 \leq j < i$ it holds that $\mathcal{J}^j \models \varphi_1$ and
10. if $\varphi = \varphi_1\mathsf{R}\varphi_2$, then $\mathcal{J} \models \varphi$ if and only if for every $i \geq 0$ such that $\mathcal{J}^i \not\models \varphi_2$ there is j such that $0 \leq j < i$ and $\mathcal{J}^j \models \varphi_1$.

As in the case of propositional and first order logic, an interpretation $\mathcal{J}$ with $\mathcal{J} \models \varphi$ for some LTL–formula φ is called a *model* of φ. The set of all models of φ is again denoted as $MD(\varphi)$. As before we define sets Φ of formulas to be satisfied by an interpretation $\mathcal{J}$ if every formula in Φ is satisfied by $\mathcal{J}$. Formally: $\mathcal{J} \models \Phi$ if and only if $I \models \varphi$ for each $\varphi \in \Phi$. The notation MD is extended to sets of LTL–formulas as before:

$$MD(\Phi) = \bigcap_{\varphi\in\Phi} MD(\varphi).$$

As before we will call a formula φ (respectively a set Φ of formulas)

- *satisfiable* if $MD(\varphi) \neq \emptyset$ (respectively $MD(\Phi) \neq \emptyset$),
- *valid* if $\mathcal{J} \models \varphi$ for each $\mathcal{J}$ (respectively if every $\varphi \in \Phi$ is valid) and
- *unsatisfiable* if $MD(\varphi) = \emptyset$ (respectively $MD(\Phi) = \emptyset$).

As one might already expect, there is also an extension of the *logical consequence* relation $\models$ known from propositional logic to LTL. Again we have

$$\Phi \models \varphi \text{ if and only if } MD(\Phi) \subseteq MD(\varphi).$$

We will also write $\varphi \models \psi$ if the set Φ only consists of the single formula φ, that is $\varphi \models \psi$ denotes $\{\varphi\} \models \psi$. The properties of $\models$ carry over from propositional logic to LTL. Furthermore we have the following lemma.

Lemma 2.3.1

Let Φ be a set of LTL–formulas and let φ be an LTL–formula.

1. Φ is unsatisfiable if and only if $\Phi \models \texttt{false}$.
2. If $\Phi \models \varphi$, then there is a finite subset $\Psi \subseteq \Phi$ such that $\Psi \models \varphi$.

In particular, testing for unsatisfiability can be accomplished by applying Lemma 2.3.1.

Corollary 2.3.1

Let Φ be a set of LTL–formulas. If $\Phi \models \texttt{false}$, then there is some finite subset $\Phi_0 \subseteq \Phi$ such that $\Phi_0 \models \texttt{false}$.

The relation $\equiv$ is again extended in the obvious way: for every pair φ_1, φ_2 of LTL–formulas we have $\varphi_1 \equiv \varphi_2$ if and only if $\varphi_1 \models \varphi_2$ and $\varphi_2 \models \varphi_1$ or equivalently if and only if $MD(\varphi_1) = MD(\varphi_2)$.

2.3.2. First Order Temporal Logic

This last section of this chapter will deal with the extension of the propositional temporal logic LTL introduced in chapter 2.3.1 to the field of first order logic. The resulting logic will consequently be denoted as FOLTL (standing for *F*irst *O*rder LTL).

Assume that a *signature* sig $= (\mathcal{X}, F, P, \alpha)$ as defined in chapter 2.2 is given. Therefore the set $\mathcal{T} = \mathcal{T}(\text{sig})$ is defined.

We will introduce the set of FoLtl–formulas stepwise.

Definition 2.3.4 (Temporal Atoms)

The set of all *temporal atoms* over sig (denoted as $\mathcal{A}_t(\text{sig})$) is defined as the smallest set of objects closed under the following rules:

1. if $\varphi \in \mathcal{A}(\text{sig})$ is a first order atomic formula, then $\varphi \in \mathcal{A}_t(\text{sig})$,
2. if φ is a temporal atom from $\mathcal{A}_t(\text{sig})$, then $\mathsf{X}\varphi$, $\mathsf{F}\varphi$ and $\mathsf{G}\varphi$ are in $\mathcal{A}_t(\text{sig})$ and
3. if φ_1 and φ_2 are in $\mathcal{A}_t(\text{sig})$, then so are $\varphi_1\mathsf{U}\varphi_2$ and $\varphi_1\mathsf{P}\varphi_2$.

The definition of *temporal literals* is very similar to the definition of temporal atoms.

Definition 2.3.5 (Temporal Literals)

The set of all *temporal literals* over sig (denoted as $\mathcal{L}_t(\text{sig})$) is defined as the smallest set of objects closed under the following rules:

1. if $\varphi \in \mathcal{A}_t(\text{sig})$ is a temporal atomic formula, then $\varphi \in \mathcal{L}_t(\text{sig})$,
2. if φ is a temporal literal from $\mathcal{L}_t(\text{sig})$, then $\mathsf{X}\varphi$, $\mathsf{F}\varphi$ and $\mathsf{G}\varphi$ are in $\mathcal{L}_t(\text{sig})$,
3. if $\varphi \in \mathcal{L}_t(\text{sig})$ is a temporal literal, then so is $\neg\varphi$ and
4. if φ_1 and φ_2 are in $\mathcal{L}_t(\text{sig})$, then so are $\varphi_1\mathsf{U}\varphi_2$ and $\varphi_1\mathsf{P}\varphi_2$.

Formulas from FoLtl are now defined as in the case of a first order logic language.

Definition 2.3.6 (First Order Linear Time Temporal Logic, e.g. [3])

The set of FoLtl–*formulas* is the smallest set of objects closed under the following rules:

1. each $\varphi \in \mathcal{L}_t(\text{sig})$ is a formula in FoLtl,

2. if φ_1 and φ_2 are formulas from FoLtl, then so are $(\varphi_1 \wedge \varphi_2)$, $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2)$,

3. if φ is a formula from FoLtl, then so are $\mathsf{X}\varphi$, $\mathsf{G}\varphi$ and $\mathsf{F}\varphi$,

4. if φ_1 and φ_2 are formulas from FoLtl, then so are $\varphi_1 \mathsf{U} \varphi_2$ and $\varphi_1 \mathsf{P} \varphi_2$ and

5. if φ is a formula from FoLtl and $x \in \mathcal{X}$, then $\forall x \varphi$ and $\exists x \varphi$ are formulas from FoLtl.

The extension of the connectives $\wedge$ and $\vee$ is extended to include arbitrary many formulas as described for first order logic formulas on page 9. We will also make use of the binding priority for the connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\rightarrow$ omitting brackets whenever this is possible.

In contrast to the propositional temporal logic Ltl one can distinguish between two kinds of symbols: *rigid* symbols and *flexible* symbols. Rigid symbols are symbols which are required to be interpreted to the same operation regardless of the point of time under consideration while flexible symbols may be interpreted as different operations at different points of time. We assume that each symbol is either flexible or rigid.

The semantics of FoLtl is described by a suitable extension of the concept of *temporal interpretations* as introduced for Ltl in chapter 2.3.1. We will follow notations from [3] which present an adaption of the so called *possible worlds semantics* which had been originally developed by Hintikka (see [81]) and Kripke (see [97]). An *interpretation* is given as a tuple

$$\mathcal{J} = (U_{\mathcal{J}}, S, s_0, \delta_1, \delta_2, w, \mathcal{I}),$$

where

- $U_{\mathcal{J}}$ is a nonempty set, called the *universe* of $\mathcal{J}$,

- S is a set of *states* (also called *possible worlds*) which contains the distinguished element s_0, the *initial state* (or *actual world*),

- $\delta_1, \delta_2 \subseteq S \times S$ are *accessibility relations*,
- $w : \mathcal{X} \to U_{\mathcal{J}}$ is an *evaluation of the variable symbols* and
- $\mathcal{I}$ is a first order interpretation for the symbols of sig which maps each symbol $\sigma \in F \cup P$ in each state s to an operation $\mathcal{I}(s, \sigma) : U_{\mathcal{J}}^{\alpha(\sigma)} \to U_{\mathcal{J}}$ (if $\sigma \in F$) or to a predicate $\mathcal{I}(s, \sigma) : U_{\mathcal{J}}^{\alpha(\sigma)} \to \{0, 1\}$ (if $\sigma \in P$). $\mathcal{I}$ is assumed to have the following properties:
  - if σ is a rigid symbol, then $\mathcal{I}(s_1, \sigma) = \mathcal{I}(s_2, \sigma)$ for every $s_1, s_2 \in S$ and
  - if σ is a flexible symbol, then there are $s_1, s_2 \in S$ such that $\mathcal{I}(s_1, \sigma) \neq \mathcal{I}(s_2, \sigma)$.

The *evaluation of terms* in such an interpretation is accomplished as expected: let $t \in \mathcal{T}(\text{sig})$ be given.

1. if $t = x \in \mathcal{X}$, then $\mathcal{J}(x) = w(x)$ and
2. if $t = f(t_1, \ldots, t_n)$ for some $f \in F$ with $\alpha(f) = n$ and $t_1, \ldots, t_n \in \mathcal{T}(\text{sig})$, then

$$\mathcal{J}(t) = \mathcal{I}(s_0, f)\,(\mathcal{J}(t_1), \ldots, \mathcal{J}(t_n))\,.$$

The interpretation of formulas is now defined similarly to the interpretation of formulas in first order logic.

- $\mathcal{J}(\texttt{true}) := 1$,
- $\mathcal{J}(\texttt{false}) := 0$ and
- if $\varphi = p(t_1, \ldots, t_n) \in \mathcal{A}(\text{sig})$ for $p \in P$ with $\alpha(p) = n$ and $t_1, \ldots, t_n \in \mathcal{T}(\text{sig})$, then

$$\mathcal{J}(\varphi) = \mathcal{J}(p(t_1, \ldots, t_n)) := \mathcal{I}(s_0, p)(\mathcal{J}(t_1), \ldots, \mathcal{J}(t_n)).$$

For the connectives $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$ which we will call *first order connectives* from now on and the quantifiers $\forall$ and $\exists$ the semantics is defined as usual. Let φ, φ_1 and φ_2 be FoLtl–formulas and let $x \in \mathcal{X}$ be a variable symbol.

- $\mathcal{J}(\neg\varphi) := 1 - \mathcal{J}(\varphi)$,
- $\mathcal{J}(\varphi_1 \wedge \varphi_2) := \min\{\mathcal{J}(\varphi_1), \mathcal{J}(\varphi_2)\}$,
- $\mathcal{J}(\varphi_1 \vee \varphi_2) := \max\{\mathcal{J}(\varphi_1), \mathcal{J}(\varphi_2)\}$,
- $\mathcal{J}(\varphi_1 \rightarrow \varphi_2) := \max\{\mathcal{J}(\neg\varphi_1), \mathcal{J}(\varphi_2)\}$,
- $\mathcal{J}(\varphi_1 \leftrightarrow \varphi_2) := \min\{\mathcal{J}(\varphi_1 \rightarrow \varphi_2), \mathcal{J}(\varphi_2 \rightarrow \varphi_1)\}$,
- $\mathcal{J}(\forall x\varphi) := \min\left\{\mathcal{J}_x^t(\varphi) \mid t \in U_{\mathcal{J}}\right\}$ and
- $\mathcal{J}(\exists x\varphi) := \max\left\{\mathcal{J}_x^t(\varphi) \mid t \in U_{\mathcal{J}}\right\}$.

where $\mathcal{J}_x^t$ emerges from $\mathcal{J}$ in a similar way as in first order logic (see page 17).

What remains to be defined is the semantics of the temporal operators. This is done via the reachability relations δ_1 and δ_2 which model the *next state*–relation (δ_1) and its transitive closure (δ_2). For modeling the semantics we will need another concept. Let $s \in S$ be any state. The interpretation $\mathcal{J}[s]$ emerges from $\mathcal{J}$ by setting its initial state (or its *actual world* which gives a better intuition in this case) from s_0 to s. Now let φ, φ_1 and φ_2 be given.

- $\mathcal{J}(\mathsf{X}\varphi) := 1$ if and only if there is $s_1 \in S$ such that $s_0\delta_1 s_1$ and $\mathcal{J}[s_1](\varphi) = 1$,
- $\mathcal{J}(\mathsf{F}\varphi) := 1$ if and only if there is $s_1 \in S$ such that $s_0\delta_2 s_1$ and $\mathcal{J}[s_1](\varphi) = 1$,
- $\mathcal{J}(\mathsf{G}\varphi) := 1$ if and only if $\mathcal{J}[s_1](\varphi) = 1$ for every $s_1 \in S$ such that $s_0\delta_2 s_1$,
- $\mathcal{J}(\varphi_1 \mathsf{U} \varphi_2) = 1$ if and only if for every $s_1 \in S$ such that $s_0\delta_2 s_1$ it holds that $\mathcal{J}[s_1](\varphi_1) = 1$ or there is $s_2 \in S$ such that $s_0\delta_2 s_2$, $s_2\delta_2 s_1$ and $\mathcal{J}[s_2](\varphi_2) = 1$ and

- $\mathcal{J}(\varphi_1 \mathsf{P} \varphi_2) = 1$ if and only if there is $s_1 \in S$ such that $s_0 \delta_2 s_1$ and $\mathcal{J}[s_1](\varphi_1) = 1$ and for each $s_2 \in S$ such that $s_0 \delta_2 s_2$ and $s_2 \delta_2 s_1$ it holds that $\mathcal{J}[s_2](\neg\varphi_2) = 1$.

As usual we will write $\mathcal{J} \models \varphi$ if $\mathcal{J}(\varphi) = 1$ and call $\mathcal{J}$ a *model*. The set of all models of φ is again denoted as $MD(\varphi)$. For sets Φ of formulas we have the obvious extension:

$$MD(\Phi) = \bigcap_{\varphi \in \Phi} MD(\varphi).$$

The notations of *satisfiability*, *validity* and *unsatisfiability* are extended in a straightforward way. To conclude the definition of the semantics of FoLtl we extend the logical consequence relation $\models$ to FoLtl–formulas and sets of FoLtl–formulas by adjusting the notations from first order logic given on page 18.

Now that both propositional and first order linear time temporal logic is defined we are ready to introduce the remaining concepts which will be the subject of the theory to be developed in this thesis, namely Logic Programming and Inductive Logic Programming.

3. Logic Programming

Contents

3.1. Predicate Logic as a Programming Language 29

3.2. The Concept of SLD–Resolution 33

3.3. Soundness and Completeness 40

This chapter briefly reviews the basic concepts of Logic Programming as introduced by Kowalski (see [94] and [93]). Logic Programming is a form of *Declarative Programming* which is a programming concept based on the philosophy that the programmer should not be concerned with the way a solution is searched for by a programming system but (s)he should be allowed to concentrate on the description of the properties of the solutions. Various realizations of declarative languages, mostly functional languages such as LISP or HASKELL, have been proposed. Logic Programming is another form of declarative programming which is concerned with describing *relationships* between objects with certain properties.

3.1. Predicate Logic as a Programming Language

The basic objects of a logic programming language are formulas of a special type, so called *clauses*.

Definition 3.1.1 (Clause, Robinson [140])

A *clause* C is a disjunction of literals l_i:

$$C = \bigvee_{i=1}^{n} l_i.$$

Assume for now that a clause C is given. C can also be seen as a set of literals $C = \{l_1, \ldots, l_n\}$. Since every literal is either an atom from $\mathcal{A}$ or the negation of an atom, we can partition C into two sets $\textsc{Pos}(C)$ and $\textsc{Neg}(C)$ containing the *positive* literals of C (i.e. the atoms[1]) and the *negative* ones (i.e. the negations of atoms):

$$C = \underbrace{\{l_i \mid i \in \{1, \ldots, n\}, l_i \in \mathcal{A}\}}_{=:\textsc{Pos}(C)} \cup \underbrace{\{l_i \mid i \in \{1, \ldots, n\}, \neg l_i \in \mathcal{A}\}}_{=:\textsc{Neg}(C)}.$$

Now we can identify several classes of clauses: C is called a

- *hornclause* if and only if $|\textsc{Pos}(C)| \leq 1$,
- *definite hornclause* if and only if $|\textsc{Pos}(C)| = 1$ and
- *unit clause* (or *fact*) if and only if $|\textsc{Pos}(C)| = 1$ and $\textsc{Neg}(C) = \emptyset$.

A *goal* (or a *query*) G is a finite sequence of atoms which are considered to be conjunctively connected: $G = G_1 \wedge \cdots \wedge G_m$. The atoms G_i are called the *subgoals of* G.

The distinguished clause which neither contains positive nor negative literals, and which is therefore represented as the set $\{\}$, is called the *empty clause* which we will denote as $\Box$[2]. The empty clause is considered equivalent to any unsatisfiable first order formula.

[1]Here we identify $\neg\neg\varphi$ and φ.

[2]Note that $\Box$ is also a goal, namely the goal which does not contain any subgoals. We will therefore also refer to $\Box$ as the *empty goal*.

The philosophy of Logic Programming is the following (see e.g. [94] and [164]):

- Statements which are known to be correct are modeled by unit clauses (facts).
- Relations between objects are modeled by definite horn clauses (which are interpreted as *rules*).
- Program calls are modeled by goals.

We will assume that every rule represents a hornclause which is *implicitly universally closed*, that is every variable symbol which occurs in a rule is assumed to be inside the scope of a universal quantifier. Formally this means that if

$$C = \varphi_1 \wedge \cdots \wedge \varphi_n \rightarrow \psi$$

is a rule which contains the variables $\{x_1, \ldots, x_m\}$ then we merely work with the formula

$$\forall x_1 \ldots \forall x_m \left(\varphi_1 \wedge \cdots \wedge \varphi_n \rightarrow \psi\right).$$

Example 3.1.1

Let sig be a signature which contains a relation symbol `is_even` with $\alpha(\texttt{is\_even}) = 1$ and function symbols `null` and `s` with $\alpha(\texttt{null}) = 0$ and $\alpha(\texttt{s}) = 1$. Then the concept of even numbers is modeled by the following set of definite horn clauses:

$$\begin{aligned} C_1 &= \texttt{is\_even}(\texttt{null}) \text{ and} \\ C_2 &= \texttt{is\_even}(x) \rightarrow \texttt{is\_even}(\texttt{s}(\texttt{s}(x))). \end{aligned}$$

Programs in a logic programming language are now given as sets of facts and rules.

Definition 3.1.2 (Logic Program, Lloyd [105])

Let sig $= (\mathcal{X}, F, P, \alpha)$ be a signature. A (logic) *program* over sig is a finite set of definite hornclauses over sig.

Example 3.1.2

Let P be the set $= \{C_1, C_2\}$ from Example 3.1.1. Then P is a logic program over the signature given there.

Due to the special structure of definite hornclauses and goals one can introduce special notations for them. Assume that C is a definite horn clause and $\textsc{Pos}(C) = \{A\}$, $\textsc{Neg}(C) = \{\neg B_1, \ldots, \neg B_n\}$ for $A, B_1, \ldots, B_n \in \mathcal{A}$. Then we have

$$\begin{aligned} C &= \{A, \neg B_1, \ldots, \neg B_n\} \\ &= A \vee \neg B_1 \vee \cdots \vee \neg B_n \\ &\equiv A \vee \neg (B_1 \wedge \cdots \wedge B_n) \\ &\equiv (B_1 \wedge \cdots \wedge B_n) \rightarrow A \\ &=: A \leftarrow B_1, \ldots, B_n. \end{aligned}$$

This can be seen as a *procedure declaration* for a procedure labeled A as described by Kowalski in [94]. The interpretation is then given as follows:

To solve A, solve B_1,B_2,...,B_n!

Now assume that G is a goal consisting of the *subgoals* G_i, that is $G = G_1 \wedge \cdots \wedge G_m$. By analogy we have

$$\begin{aligned} \neg G &= \neg (G_1 \wedge \cdots \wedge G_m) \\ &\equiv \neg G_1 \vee \cdots \vee \neg G_m \\ &=: \leftarrow G_1, \ldots, G_m. \end{aligned}$$

Using the procedural interpretation from above we can see a goal as the statement

Solve G_1,G_2,...,G_m!

Now if one wants to *run* a logic program a goal G is added to the program and a theorem proving procedure tests if the goal is a logical consequence of the program under consideration. The task how this is achieved is the subject of the next section of this chapter.

3.2. The Concept of SLD–Resolution

This section will be concerned with a brief explanation of the concept of *SLD–resolution* (originally introduced in [95]) which is a theorem proving procedure designed to handle (definite) hornclauses. It is a refinement of the classical *Resolution* procedure introduced by Robinson (see [140]). Since there are many excellent texts on theorem proving in general (and especially on the topic of logic programming) the discussion will be rather short. We refer the interested reader to the literature (see [31], [8], [7], [105] and [126]).

The key result for understanding SLD–Resolution is given by the following lemma which is a special case of Proposition 3.1 from [105].

Lemma 3.2.1 (Lloyd [105])
Let P be a logic program and let G be a goal. Then $P \models G$ if and only if $P \cup \{\leftarrow G\} \models \Box$.

This lemma can also be seen as an easy consequence of Theorem 2.2.3. So if $G_1 \wedge \cdots \wedge G_m$ is a logical consequence of P one only has to deduce the empty clause from $P \cup \{\neg(G_1 \wedge \cdots \wedge G_m)\}$. This is achieved by application of the principle of SLD–resolution which we will define now.

Several other approaches for implementing logical programming languages have been proposed. In principle it is possible to take *any* complete proof procedure for first order logic in order to achieve this goal. But due to implementation difficulties and performance problems one concentrates on refutation–complete calculi. Popular approaches are based on tableaux techniques (see e.g. [67] and [4] for calculi for definite logic programs and [19] and [130] for calculi for disjunctive logic programming languages) and the Model Elimination Technique as described by Loveland in [107] and [108] (see e.g. [70]). However, the

SLD–resolution approach has been the first method to be implemented in PROLOG systems and is therefore still a dominant procedure in logic programming systems. Although SLD–Resolution is sound and refutation–complete many implementations omit certain operations which are necessary in order to guarantee these properties. In particular, the occur–check which is necessary during the unification process is a very expensive operation and therefore many implementations skip this check. Although there are programs which allow skipping this check (see [9]) omitting the check results in losing the property of soundness (see e.g. [112]). Another expensive operation is the breadth–first search strategy which is necessary in order to guarantee refutation–completeness. Most PROLOG systems simply carry out *one* inference step (namely the first one which is possible) without trying other steps which might be applicable. This might result in non–halting derivations which have trivial solutions.

In order to reason about instantiations of formulas one has to define a suitable concept of *substitution*. Intuitively a substitution replaces variables by terms. Formally a substitution is defined as a certain type of homomorphism on terms and formulas.

Definition 3.2.1 (Substitution, Robinson [140])
A *substitution* is a mapping $\sigma : \mathcal{X} \to \mathcal{T}$ such that $\{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite.

Since the set of variables which are changed by the substitution σ is required to be finite, we can write down substitutions by stating which variables are replaced by which terms and omitting the variables which remain unchanged. The set of all these variables will be called the *domain* of the substitution σ and will be denoted as $\textsc{Dom}(\sigma)$. Assume that $\textsc{Dom}(\sigma) = \{x \mid \sigma(x) \neq x\} = \{x_{i_1}, \dots, x_{i_n}\}$ and that $\sigma\left(x_{i_j}\right) = t_j \in \mathcal{T}$ for $j = 1, \dots, n$. Then σ will be identified by the set of *bindings*

$$\left\{\frac{x_{i_1}}{t_1}, \dots, \frac{x_{i_n}}{t_n}\right\} = \left\{\frac{x_{i_1}}{\sigma\left(x_{i_1}\right)}, \dots, \frac{x_{i_n}}{\sigma\left(x_{i_n}\right)}\right\}$$

where each of the n bindings $\frac{x_{i_j}}{t_j}$ denotes the substitution σ_j with

$$\sigma_j(x) = \begin{cases} t_j & \Leftrightarrow x = x_{i_j} \\ x & \Leftrightarrow \text{else} \end{cases}$$

Substitutions are extended to terms and formulas by defining homomorphic extensions:

$$\begin{aligned}
\sigma(f(t_1,\ldots,t_n)) &= f(\sigma(t_1),\ldots,\sigma(t_n)),\\
\sigma(p(t_1,\ldots,t_n)) &= p(\sigma(t_1),\ldots,\sigma(t_n)),\\
\sigma(\neg\varphi) &= \neg\sigma(\varphi),\\
\sigma(\varphi_1 \wedge \varphi_2) &= \sigma(\varphi_1) \wedge \sigma(\varphi_2),\\
\sigma(\varphi_1 \vee \varphi_2) &= \sigma(\varphi_1) \vee \sigma(\varphi_2),\\
\sigma(\varphi_1 \rightarrow \varphi_2) &= \sigma(\varphi_1) \rightarrow \sigma(\varphi_2),\\
\sigma(\varphi_1 \leftrightarrow \varphi_2) &= \sigma(\varphi_1) \leftrightarrow \sigma(\varphi_2),\\
\sigma(\forall x\varphi) &= \forall x\sigma(\varphi) \text{ and}\\
\sigma(\exists x\varphi) &= \exists x\sigma(\varphi).
\end{aligned}$$

Substitutions can be *composed* in order to build *complex* substitutions from *simpler* ones. This will be interesting for us in the following part of this section when we will define the result computed by a logic program P given a goal G as input.

Definition 3.2.2 (Composition of Substitutions, Robinson [140])
Let $\sigma_1 = \left\{\frac{x_1^{(1)}}{t_1^{(1)}},\ldots,\frac{x_n^{(1)}}{t_n^{(1)}}\right\}$ and $\sigma_2 = \left\{\frac{x_1^{(2)}}{t_1^{(2)}},\ldots,\frac{x_m^{(2)}}{t_m^{(2)}}\right\}$ be substitutions. Then the substitution $\sigma_1 \circ \sigma_2$ (the *composition* of σ_1 and σ_2) is defined as follows:

$$\sigma_1 \circ \sigma_2 \quad = \quad \left\{\frac{x_1^{(1)}}{\sigma_2\left(t_1^{(1)}\right)},\ldots,\frac{x_n^{(1)}}{\sigma_2\left(t_n^{(1)}\right)},\frac{x_1^{(2)}}{t_1^{(2)}},\ldots,\frac{x_m^{(2)}}{t_m^{(2)}}\right\}$$

$$\setminus \left(\left\{ \frac{x_i^{(1)}}{\sigma_2\left(t_i^{(1)}\right)} \mid x_i^{(1)} = \sigma_2\left(t_i^{(1)}\right) \right\} \cup \left\{ \frac{x_j^{(2)}}{t_j^{(2)}} \mid x_j^{(2)} \in \left\{ x_1^{(1)}, \ldots, x_n^{(1)} \right\} \right\} \right)$$

That is in order to compute the composition of σ_1 and σ_2 one first applies σ_2 to the terms which σ_1 replaces for the variables in its domain, then adds the bindings of σ_2 (in order to add the elements which are not yet in the domain of σ_1) and then reduces the resulting set by removing identical bindings and such bindings which won't have any effect.

We will identify a distinguished element ε as the substitution which does not contain any binding and so will have no effect on the objects to which it is applied. This element is given as $\varepsilon = \emptyset$ and is denoted as the *empty substitution*. The composition operation $\circ$ enjoys the following properties (Proposition 4.1 from [105]):

Lemma 3.2.2 (Properties of Substitutions, Lloyd [105])

Let σ_1, σ_2 and σ_3 be substitutions, let t be a term and let φ be a formula. Then

1. $\sigma_1 \circ \varepsilon = \varepsilon \circ \sigma_1 = \sigma_1$,
2. $\sigma_2(\sigma_1(t)) = (\sigma_1 \circ \sigma_2)(t)$,
3. $\sigma_2(\sigma_1(\varphi)) = (\sigma_1 \circ \sigma_2)(\varphi)$ and
4. $(\sigma_1 \circ \sigma_2) \circ \sigma_3 = \sigma_1 \circ (\sigma_2 \circ \sigma_3)$.

Let φ_1 and φ_2 be arbitrary formulas from $\mathcal{F}$. We say that φ_1 and φ_2 are *variants* if there are substitutions σ_1 and σ_2 such that $\sigma_1(\varphi_1) = \varphi_2$ and $\sigma_2(\varphi_2) = \sigma_1$.

Now let t_1, t_2 be terms and let φ_1, φ_2 be literals. Substitutions σ which yield syntactically identical objects, i.e. substitutions σ such that $\sigma(t_1) = \sigma(t_2)$ or $\sigma(\varphi_1) = \sigma(\varphi_2)$ play an important role.

Definition 3.2.3 (Unification)

Let t_1 and t_2 be terms, let φ_1 and φ_2 be literals. t_1 and t_2 (respectively φ_1 and φ_2) are said to be *unifiable* if and only if there is a substitution σ such that $\sigma(t_1) = \sigma(t_2)$ (respectively $\sigma(\varphi_1) = \sigma(\varphi_2)$). σ is called a *unifier.*

Some unifiers can be characterized as unifiers which only change the objects to be unified *as much as necessary.*

Definition 3.2.4 (Most General Unifier)

Let t_1 and t_2 be terms and let φ_1 and φ_2 be literals. A unifier σ for t_1 and t_2 (respectively for φ_1 and φ_2) is called a *most general unifier* (or *mgu*) if and only if for every unifier σ_1 for t_1 and t_2 (respectively φ_1 and φ_2) there is a substitution σ_2 such that $\sigma = \sigma_1 \circ \sigma_2$. We will denote this by writing $\sigma = \mathrm{mgu}(t_1, t_2)$ (respectively $\sigma = \mathrm{mgu}(\varphi_1, \varphi_2)$).

Having defined which properties a most general unifier satisfies it is necessary to ensure that in the case of unifiable objects there is indeed an algorithm which can compute such a substitution. The following lemma from [140] ensures this.

Lemma 3.2.3 (Unification Theorem, Robinson [140])

Let t_1 and t_2 be terms and let φ_1 and φ_2 be literals. If t_1 and t_2 (respectively φ_1 and φ_2) are unifiable, then there exists a mgu of t_1 and t_2 (respectively φ_1 and φ_2) which is uniquely determined up to renaming of variables and which can be effectively computed.

Several algorithms have been proposed for computing most general unifiers. The first and also most simple one was presented by Robinson in 1965 (see [140]) which has the drawback that its worst case runtime is exponential in the length of the objects to be identified. It has also been shown that the unification problem is solvable in linear time by Paterson and Wegman which presented an algorithm which operates on directed acyclic graphs representing the objects (see [129]). However, due to its simplicity another algorithm introduced by Martelli and Montanari in 1982 (see [113]) is used very often in Logic Programming systems. We will refine this algorithm in order to be capable of handling temporal objects in a later chapter.

Now let $P = \{P_1, \ldots, P_n\}$ be a logic program. Furthermore assume that $G = G_1 \wedge \cdots \wedge G_m$ is a goal. Due to the special structure of definite horn clauses we can assume that each P_i can be written as

$$P_i = A_i \leftarrow B_1^{(i)}, \ldots, B_{n_i}^{(i)}$$

for some n_i and $A_i, B_1^{(i)}, \ldots, B_{n_i}^{(i)} \in \mathcal{A}$. Assume that i and j are such that G_j and A_i are unifiable with $\sigma = \mathrm{mgu}(A_i, G_j)$. The *resolvent* (or *SLD–resolvent*) of G and P_i with respect to σ is the goal

$$\begin{array}{rl} & \sigma\left(G_1 \wedge \cdots \wedge G_{j-1} \wedge B_1^{(i)} \wedge \ldots, B_{n_i}^{(i)} \wedge G_{j+1} \wedge \cdots \wedge G_m\right) \\ = & \sigma\left(G_1\right) \wedge \cdots \wedge \sigma\left(G_{j-1}\right) \wedge \sigma\left(B_1^{(i)}\right) \wedge \cdots \wedge \sigma\left(B_{n_i}^{(i)}\right) \wedge \sigma\left(G_{j+1}\right) \wedge \cdots \wedge \sigma\left(G_m\right) \\ =: & \left(G_1 \wedge \cdots \wedge G_{j-1} \wedge B_1^{(i)} \wedge \cdots \wedge B_{n_i}^{(1)} \wedge G_{j+1} \wedge \cdots \wedge G_m\right)\sigma \end{array}$$

Definition 3.2.5 (SLD–derivation–step, Kowalski and Kuehner [95])
Let P be a logic program and let $G = G_1 \wedge \cdots \wedge G_m$ be a goal. An *SLD–derivation–step* is a sequence of actions carried out as follows:

1. A subgoal G_i is chosen.
2. A clause $A \leftarrow B_1, \ldots, B_n$ from P is chosen such that A and G_i are unifiable. Let σ be a most general unifier for A and G_i.
3. The SLD–resolvent G' of G and $A \leftarrow B_1, \ldots, B_m$ with respect to σ is constructed.

Given P we will write $G \vdash_{\text{Res}} G'$ if there is an SLD–derivation–step yielding G' from G. The relation $\vdash$ is then defined as the reflexive–transitive closure of $\vdash_{\text{Res}}$:

$$\begin{array}{rcl} G \overset{0}{\vdash} G' & :\Leftrightarrow & G = G', \\ G \overset{1}{\vdash} G' & :\Leftrightarrow & G \vdash_{\text{Res}} G' \text{ and} \\ G \overset{n+1}{\vdash} G' & :\Leftrightarrow & \text{there is } \bar{G} \text{ such that } G \overset{n}{\vdash} \bar{G} \text{ and } \bar{G} \overset{1}{\vdash} G' \end{array}$$

So we can define $G \vdash G'$ if there is some $n \geq 0$ such that $G \overset{n}{\vdash} G'$. In other words $\vdash = \vdash^*_{\text{Res}}$.

An *SLD–derivation* of G from P is a sequence $(G_i)_{i \geq 0}$ such that $G_0 = G$ and for each i there is a clause from P which can be used in order to carry out an SLD–derivation–step yielding G_{i+1}.

An SLD–derivation $(G_i)_{i \geq 0}$ of G from P which is of finite length, say $(G_i)_{i=1}^n$ is called an *SLD–derivation of G_n from P given input G.*

An SLD–derivation $(G_i)_{i=1}^n$ of G_n from P given input G_0 is called *successful* if $G_n = \Box$.

In the case that there is a successful SLD–derivation of $\Box$ from P given input G we will write $P \vdash G$, otherwise we will write $P \not\vdash G$. Furthermore we will call successful SLD–derivations of $\Box$ from P given input G *SLD–refutations* of $\leftarrow G$ from P. Sometimes we will also refer to SLD–derivations of goals from P given input G as SLD–derivations of $P \cup \{\leftarrow G\}$.

So given a program P and a goal G one can construct *all* possible derivations of $P \cup \{\leftarrow G\}$ and check if there is a refutation of $P \cup \{\leftarrow G\}$. If such a refutation exists, then it can be found by breadth–first–search. If no such refutation exists, then in general there is no way to detect this since the fragment of clausal logic is undecidable (see [142]).

Example 3.2.1

Again consider the program from Examples 3.1.1 and 3.1.2. Assume that the goal G is given by $G =$ `is_even(s(s(s(s(null)))))`. Then an SLD–refutation of $P \cup \{\leftarrow G\}$ can be visualized as depicted in Figure 3.1 where an arrow between two goals means that the

$$\begin{tikzcd}
\texttt{is\_even(s(s(s(s(null)))))} \arrow[d] \\
\texttt{is\_even(s(s(null)))} \arrow[d] \\
\texttt{is\_even(null)} \arrow[d] \\
\square
\end{tikzcd}$$

Figure 3.1.: SLD–refutation

second goal is a resolvent of the first one (with respect to a suitable mgu).

3.3. Soundness and Completeness

As we have seen computations by logic programs are carried out by giving a goal G to a program P and proving that the set $P \cup \{\leftarrow G\}$ is unsatisfiable, that is proving that $P \cup \{\leftarrow G\} \vdash \square$ holds. The *result* of such a computation is given by the substitution which emerges from composing the unifiers used in this refutation. Let σ be this substitution. We first state the following lemma:

Lemma 3.3.1 (Soundness, Apt and v. Emden [10])
The SLD–resolution rule is sound. That is if $P \vdash \varphi$, then $P \models \varphi$.

This lemma is proved directly in [10] but it is also a consequence of the soundness of the general resolution rule presented in [140].

So assume that $P \cup \{\leftarrow G\} \vdash \square$. Then we have $P \cup \{\leftarrow G\} \models \square$ and due to Lemma 3.2.1 we have $P \models G$. Since G is a goal, we have $G = \leftarrow G_1, \dots, G_m$ for atoms G_i and therefore $P \models G_1 \wedge \dots \wedge G_m$.

We introduce two concepts of substitutions which will turn out to be useful.

Definition 3.3.1 (Answer, Lloyd [105])
Let P be a logic program and let $G = G_1, \dots, G_m$ be a goal.

1. An *answer* for $P \cup \{\leftarrow G\}$ is a substitution for the variables occurring in G.
2. A *correct answer* for $P \cup \{\leftarrow G\}$ is an answer σ such that $P \models (G_1 \wedge \cdots \wedge G_m)\,\sigma$.
3. A *computed answer* for $P \cup \{\leftarrow G\}$ is the composition of the most general unifiers used in a refutation of $P \cup \{\leftarrow G\}$.

So a computed answer for $P \cup \{\leftarrow G\}$ can be seen as a result computed by the program P given a goal G as *input.* Indeed, this interpretation of logic programs and goals is an adequate way to carry out computations as the next theorem shows.

Theorem 3.3.1 (Refutation–Completeness, v. Emden and Kowalski [164]) Let P be a logic program and let G be a goal.

1. Every computed answer for $P \cup \{\leftarrow G\}$ is a correct answer for $P \cup \{\leftarrow G\}$.
2. For every correct answer σ for $P \cup \{\leftarrow G\}$ there is a computed answer σ_1 for $P \cup \{\leftarrow G\}$ and a substitution σ_2 such that $\sigma = \sigma_1 \circ \sigma_2$.

So if an answer is computed by using SLD–resolution this answer is a correct solution of the problem modeled by the program P under consideration (soundness). Additionally it is possible to compute any answer which can be instantiated to a correct one (completeness). Furthermore it is possible to model every *computable* function by a suitable logic program (see [5]). So predicate logic can indeed be seen as an adequate formalism for computation.

4. Inductive Logic Programming

Contents

4.1. The basic Framework 43

4.2. Generalization and Specialization 46

4.3. Refinement Operators and their Properties 47

This chapter will be concerned with a brief introduction of the concepts from *Inductive Logic Programming* (or *ILP* for short) which we will extend to the temporal logic to be defined in the next part of this thesis. Therefore we will first present a description of the tasks which ILP systems have to perform. After this we will introduce several important concepts from lattice theory which will be used throughout the next chapters.

4.1. The basic Framework

Inductive Logic Programming is concerned with *synthesizing* general rules from examples. Henceforth it is a special case of the theory of *algorithmic learning.* In general algorithmic learning is a generic term for every theory which is concerned with determining *explanations* for certain phenomena. In ILP, the objects (or *concepts*) to be learned are *logic programs.* The *hints* about the concept to be learned are given by sets $\mathcal{E}^+$ and $\mathcal{E}^-$ consisting of *ground atoms.* These sets are considered to be *examples* for the (unknown) program P to be learned. $\mathcal{E}^+$ contains the *positive examples* and $\mathcal{E}^-$ contains the *negative* ones. The interpretation of positive end negative examples is then given by

- $P \models e_+$ for every $e_+ \in \mathcal{E}^+$ and
- $P \not\models e_-$ for every $e_- \in \mathcal{E}^-$.

The problem setting which we will use here is usually referred to as the *normal problem setting for ILP*[1]. Some properties of a program P and sets $\mathcal{E}^+$ and $\mathcal{E}^-$ can be defined formally as follows (following [126]).

Definition 4.1.1 (Properties of Programs, e.g. [126])
Let P be a logic program and let $\mathcal{E}^+$ and $\mathcal{E}^-$ be (finite) sets of ground atoms. P is called

- *complete* wrt. $\mathcal{E}^+$ if $P \models e_+$ for every $e_+ \in \mathcal{E}^+$,
- *consistent* wrt. $\mathcal{E}^-$ if $P \cup \{\neg e_- \mid e_- \in \mathcal{E}^-\} \not\models \Box$ and
- *correct* wrt. $\mathcal{E}^+$ and $\mathcal{E}^-$ if P is complete wrt. $\mathcal{E}^+$ and consistent wrt. $\mathcal{E}^-$.

Additionally the following definitions allow a closer classification of programs relative to given sets of examples.

Definition 4.1.2 (Further Properties, e.g. [126])
Let P be a logic program and let $\mathcal{E}^+$ and $\mathcal{E}^-$ be sets of positive and negative examples. P is called

- *too strong wrt.* $\mathcal{E}^-$ if P is not consistent wrt. $\mathcal{E}^-$,
- *too weak wrt.* $\mathcal{E}^+$ if P is not complete wrt. $\mathcal{E}^+$,
- *overly general wrt.* $\mathcal{E}^+$ *and* $\mathcal{E}^-$ if P is complete wrt. $\mathcal{E}^+$ and not consistent wrt. $\mathcal{E}^-$ and

[1]In contrast to the *normal setting* the *nonomontonic setting for ILP* has been defined (see [79]).

- *overly specific wrt.* $\mathcal{E}^+$ *and* $\mathcal{E}^-$ if P is consistent wrt. $\mathcal{E}^-$ and not complete wrt. $\mathcal{E}^+$.

The task, which an ILP system has to perform is to find (or *synthesize*) a program P which is correct wrt. given sets $\mathcal{E}^+$ and $\mathcal{E}^-$.

Usually this *normal setting* is extended in a way that the usage of *background knowledge* is possible. Background knowledge is given as a finite set $\mathcal{B}$ of clauses which are interpreted as *rules which are known to be correct.* The program P to be synthesized is now required to have the following properties:

- $P \cup \mathcal{B} \models e_+$ for every $e_+ \in \mathcal{E}^+$ and
- $P \cup \mathcal{B} \not\models e_-$ for every $e_- \in \mathcal{E}^-$.

It is possible to distinguish between several ways the examples are presented to an ILP system.

Batch / Incremental In a *batch learning system* the examples from $\mathcal{E}^+$ and $\mathcal{E}^-$ are given to the system at the beginning of the learning task. In contrast, an *incremental learning system* receives the examples at different points of time. An example for a batch learning system is FOIL (see [137]).

Top down / Bottom up The distinction between top down and bottom up systems comes from the *direction* in which the system searches for a correct program. While in a top down system an overly general set $P \cup \mathcal{B}$ is *specialized*, in a bottom up system an overly specific set $P \cup \mathcal{B}$ is *generalized.* Generalization and specialization will be treated in depth in a later chapter taking temporal literals and clauses into account.

Interactive / Noninteractive An interactive system is capable of interacting with the user. Therefore such a system can ask if some assumptions it has generated while searching for a correct program are fulfilled or not. This allows the generation of *better* programs.

In all cases, the program which is generated from the sets $\mathcal{E}^+$, $\mathcal{E}^-$ and $\mathcal{B}$ cannot be guaranteed to be exactly the program which the person who has generated the examples has in mind. Rather it is (in most cases[2]) possible to construct a program which is correct with respect to examples seen so far and which has a very high probability of being correct for *other examples* which have not yet been seen.

4.2. Generalization and Specialization

In order to modify a logic program to fit its specification one has to *refine* the program by manipulating clauses in order to imply *more* or *less* atoms. The basis of all refinement operations is some fixed *generality ordering*, mostly the *subsumption ordering* which is both relatively powerful and still decidable (in contrast to the ordering induced by logical implication which in general is undecidable). We will here restrict our attention on refinement operations on pairs of clauses. Assume that a *quasi ordering* $\preceq$[3] on the set of clauses over some signature is given. Further assume that C_1 and C_2 are clauses. A clause C is then called (following [133])

- a *generalization* of C_1 and C_2 if $C \succeq C_1$ and $C \succeq C_2$ and
- a *specialization* of C_1 and C_2 if $C \preceq C_1$ and $C \preceq C_2$.

Certain specializations and generalizations are of special interest in ILP. A clause C is called

- a *least generalization* of C_1 and C_2 if C is a generalization of C_1 and C_2 and for every generalization D of C_1 and C_2 it holds that $D \succeq C$ and

[2]Note that there may exist (nontrivial) $\mathcal{E}^+$ and $\mathcal{E}^-$ such that no correct P may exist, see [126] for a proof of this.

[3]$\preceq$ is called a *quasi–ordering* if $\preceq$ is reflexive and transitive. Given $\preceq$ the notation $\succeq$ will be used as expected. We will write $\approx$ if both $\preceq$ and $\succeq$ holds and $\prec$ (resp. $\succ$) if $\preceq$ (res. $\succeq$) and not $\approx$ holds for pairs of objects of the underlying set.

- a *greatest specialization* of C_1 and C_2 if C is a specialization of C_1 and C_2 and for every specialization D of C_1 and C_2 it holds that $C \succeq D$.

A pair $(S, \preceq)$ consisting of a nonempty set S and a quasi–ordering $\preceq$ is called a *lattice* if for every pair x_1, x_2 of elements from S there exists a least generalization and a greatest specialization wrt. $\preceq$ *in* S.

Assuming that the chosen ordering $\preceq$ yields a lattice structure[4], operations for refinement can be implemented in two ways:

Upward Refinement Given C_1 and C_2 construct a least generalization of C_1 and C_2 wrt. $\preceq$.

Downward Refinement Given C_1 and C_2 construct a greatest specialization of C_1 and C_2 wrt. $\preceq$.

4.3. Refinement Operators and their Properties

As we have described in chapter 4.2, specialization and generalization are central operations which any ILP system has to perform. An algorithm which is capable of generalizing and/or specializing clauses is called a *refinement operator*. Consequently one can distinguish between *Upward Refinement Operators* and *Downward Refinement Operators* depending on the *direction* in which the refinement is performed, that is depending on the question whether it constructs a generalization or a specialization of the input clauses.

Formally we assume that a generality ordering $\preceq$ (a quasi ordering) is given as described above. A *downward refinement operator* is a function ρ_d mapping clauses to sets of clauses such that for every clause C it holds that

$$\rho_d(C) \subseteq \{D \mid C \succeq D\}.$$

[4]In chapter 6 we will see that there are indeed orderings which have this property.

Consequently an *upward refinement operator* is given as a function ρ_u mapping clauses to sets of clauses such that for each clause C it holds that

$$\rho_u(C) \subseteq \{D \mid D \succeq C\}.$$

Let ρ be any refinement operator (upward or downward) and let C be a clause. According to the notational conventions from [126] we define *1–step–refinement*, *n–step–refinements* and *refinements* as follows:

$$\begin{aligned} \rho^1(C) &= \rho(C) \quad \text{(1–step–refinement)},\\ \rho^n(C) &= \rho(\rho^{n-1}(C)), n > 1 \quad (n\text{–step–refinement) and}\\ \rho^*(C) &= \bigcup_{i \geq 1} \rho^i(C) \quad \text{(refinement)} \end{aligned}$$

ρ is called

- *locally finite* if for every C the set $\rho(C)$ is finite and computable,
- *complete* if for every pair C_1, C_2 such that $C_1 \succ C_2$ (or $C_1 \prec C_2$ for upward refinement operators) there is an element $C \in \rho^*(C_1)$ such that $C \approx C_2$,
- *proper* if for every C it holds that $\rho(C) \subseteq \{D \mid D \succ C\}$ (or $\rho(C) \subseteq \{D \mid D \prec C\}$) and
- *ideal* if ρ is locally finite, complete and proper.

Ideality seems to be a desirable property of refinement operators. However, in general it is not possible to guarantee the existence of such ideal refinement operators (see for example [162]).

Part II.

First Order Inductive Temporal Logic Programming

5. The Programming Language Prolog(+T)

Contents

5.1. Syntax of Prolog(+T) 52

5.1.1. Terms in Prolog(+T) 52

5.1.2. Facts 55

5.1.3. Rules 55

5.1.4. Programs 57

5.1.5. Queries 59

5.1.6. The Relation of Prolog(+T) to other Temporal Logic programming Languages 60

5.2. Declarative Semantics of Prolog(+T) 65

5.2.1. Preliminaries 65

5.2.2. Reduction of Literals 74

5.2.3. Semantics for programs 81

5.3. A Saturation–based temporal Proof Procedure 86

5.3.1. Tableaux Rules 87

5.3.2. Soundness and Completeness Issues 104

We will now introduce the programming language of interest. This language will be similar to Prolog (see [154] for an introduction) but will allow the usage of temporal operators. Therefore we will call it Prolog(+T). Prolog(+T) will allow list processing (as Prolog does) but *constraints* have not yet been integrated.

5.1. Syntax of Prolog(+T)

5.1.1. Terms in Prolog(+T)

Since Prolog(+T) is essentially a logic programming language which is enriched with the temporal operators X, G, F, U and P, the basic concepts are the same as described in chapter 3.1. However, to become a *practically usable* language, the usage of terms has to be simplified in order to allow more comfortable manipulating techniques. In particular, Prolog(+T) allows *lists* as terms which is probably the most comfortable feature of Prolog.

Definition 5.1.1 (Prolog(+T)–terms)
Let sig $= (\mathcal{X}, F, P, \alpha)$ be a signature. The set of Prolog(+T)–terms over sig is defined to be the smallest set which is closed under the following rules:

1. _ is a Prolog(+T)–term,
2. every string representing an *integer* is a Prolog(+T)–term,
3. every term $t \in \mathcal{T}(\mathrm{sig})$ is a Prolog(+T)–term,
4. if $t_1, \ldots, t_n$ is a finite (possibly empty) sequence of Prolog(+T)–terms, then $[t_1, \ldots, t_n]$ is a Prolog(+T)–term and
5. if t is a Prolog(+T)–term and $t_1, \ldots, t_n$ is a finite (possibly empty) sequence of Prolog(+T)–terms, then $[t|t_1, \ldots, t_n]$ is a Prolog(T)–term.

We will make use of the following *convention*: Variable identifiers will start with upper case letters while other identifiers such as function symbols and predicate symbols have to start with lower case letters.

The term `_` is referred to as an *anonymous variable*. Anonymous variables are a concept which is recently used in PROLOG–programs in order to define predicates which have variables as parameters which are not used in the definition. An example might be the definition of the nonnegative subtraction function:

$$\mathrm{nndiff}(x,y) = \begin{cases} x - y & \Leftrightarrow x \geq y \\ 0 & \text{else} \end{cases}.$$

If x is a variable symbol and `0` is a term representing the natural number 0, then the following 3–ary predicate might be *part* of the definition of the function nndiff : $\mathbb{N}^2 \rightarrow \mathbb{N}$:

$$\mathtt{p}_{\mathrm{nndiff}}(\mathtt{0}, \mathtt{X}, 0).$$

This models the following part of the definition of nndiff: $\mathrm{nndiff}(0, x) = 0$ for every value of x. Consequently, the value of x itself does not play any role in this case. So the occurrence of x can be replaced by the anonymous variable `_` here which yields the predicate

$$\mathtt{p}_{\mathrm{nndiff}}(\mathtt{0}, \_, 0).$$

In general, anonymous variables might be used whenever one is only concerned with proving that a solution *exists* without being interested in the actual value of this solution.

Integers are included in order to yield more comfortable programming facilities. It is obvious that integers are not essential for the completeness of the language since it is possible to define the natural numbers $\mathbb{N}$ (and therefore the integers $\mathbb{Z}$) in terms of a constant symbol `null` and an unary function `s` realizing the successor function.

The terms built up using the last two points from the above definition are referred to as *lists*. They represent *collections* of elements. The term [] represents the so called *empty list* which does not contain any elements. If a list can be described by completely writing down all its elements, then it might be realized as a list of the form $[t_1, \dots, t_n]$. Note that the empty list is just a special case of such a list and also note that some of the elements $t_1, \dots, t_n$ might again be lists since the definition of PROLOG(+T)–terms is recursive. So the following construct is a well–formed PROLOG(+T)–term[1]:

$$[\mathtt{f(X)}, \mathtt{g(f(X), a)}, [], [\mathtt{Y}|\mathtt{a}, \mathtt{b}, \mathtt{c}]].$$

In lists of the form $t = [\bar{t}|t_1, \dots, t_n]$, the term $\bar{t}$ is usually referred to as the *head* of the list while the list $[t_1, \dots, t_n]$ is referred to as the *tail* of the list.

For all kinds of PROLOG(+T)–terms t we define the set $\textsc{Var}(t)$ to contain all variables occurring in t. Formally this is accomplished as follows.

Definition 5.1.2 (Variables in PROLOG(+T)–terms)
Let t be a PROLOG(+T)–term. Then the set $\textsc{Var}(t)$ of *variables* of t is defined as follows:

1. if $t = \_$ or $t = []$, then $\textsc{Var}(t) = \emptyset$,
2. if $t = \mathtt{X}$ for some variable identifier $\mathtt{X}$, then $\textsc{Var}(t) = \{\mathtt{X}\}$,
3. if $t = \mathtt{f}(t_1, \dots, t_n)$ or $t = [t_1, \dots, t_n]$, then $\textsc{Var}(t) = \bigcup_{i=1}^{n} \textsc{Var}(t_i)$ and
4. if $t = [\bar{t}|t_1, \dots, t_n]$, then $\textsc{Var}(t) = \textsc{Var}(\bar{t}) \cup \bigcup_{i=1}^{n} \textsc{Var}(t_i)$.

A variable is said to *occur* in a term t if and only if it is a member of the set $\textsc{Var}(t)$.

Anonymous variables and the list concept are only introduced in order to improve the usability of PROLOG(+T) for practical applications. It is possible to show that omitting these concepts does not effect the expressivity of the language. Therefore we will not

[1]Provided that the signature of interest does contain definitions for the symbols used in this term.

deal with lists and anonymous variables in proofs of properties in the sequel.

5.1.2. Facts

As in PROLOG, PROLOG(+T) allows modeling relationships between objects as *rules* and *facts* (which are a special case of rules, namely rules without premises). Roughly speaking, rules are models of inferences which may be carried out. In general such inferences may only be applicable if some *premises* are fulfilled. For facts no such premises are present: any fact is interpreted as *something which is true without having to be proved.* As in PROLOG, facts in PROLOG(+T) are equivalent to atomic formulas.

Definition 5.1.3

A PROLOG(+T)–fact is a string of the form

$$\varphi.,$$

where $\varphi \in \mathcal{A}_t(\mathrm{sig})$ (for some suitable signature sig) is an atomic formula of FOLTL.

Note the symbol . at the end of the definition of a fact. This comes from the circumstance that facts are special cases of rules.

5.1.3. Rules

As described above, *rules* in general model allowed inferences which may be applied if some premises are known to be fulfilled. Rules are (as described in chapter 3.1) modeled as clauses. However, we will not restrict ourselves on (definite) hornclauses here.

Definition 5.1.4 (Rules in PROLOG(+T))

Let $\mathrm{sig} = (\mathcal{X}, F, P, \alpha)$ be a signature. A PROLOG(+T)–rule is either a PROLOG(+T)–

fact or a statement of the form

$$\psi :- \varphi_1, \ldots, \varphi_n.$$

for $\psi \in \mathcal{A}_t(\text{sig})$ and $\varphi_1, \ldots, \varphi_n \in \mathcal{L}_t(\text{sig})$. The atom ψ is called the *head* of the rule while the set $\{\varphi_1, \ldots, \varphi_n\}$ is called the *tail* of the rule.

So the tail of a rule describes the premises which have to be proved while the head describes the proposition which can be inferred. In general the tail may contain *negative* literals, that is $\varphi_i \in \mathcal{L}_t(\text{sig}) \setminus \mathcal{A}_t(\text{sig})$ is permitted. This has both advantages and disadvantages. On the one hand, allowing negated premises yields better readable, more elegant and shorter programs, on the other hand the definition of the semantics of programs containing rules with negated premises is more complicated to describe. We will see how this can be overcome in a later section of this chapter.

For reasons of readability we will make the following *convention*: if

$$\psi :- \varphi_1, \ldots, \varphi_n.$$

is a PROLOG(+T)–rule and φ_i is from $\mathcal{L}_t(\text{sig}) \setminus \mathcal{A}_t(\text{sig})$, that is $\varphi_i = \neg\varphi$ for some $\varphi \in \mathcal{L}_t(\text{sig})$, then we will write $\varphi_i = \texttt{not}(\varphi)$ instead. Another reason for using the above notation is the fact that we will not be dealing with *classical* negation but merely with *negation as failure* (or better: with an adaption of this negation–approach). Intuitively $\texttt{not}(\varphi)$ is assumed to be a logical consequence of a program P if φ cannot be *proved* from P in a finite number of proof steps or equivalently if every attempt to prove φ from P fails after a finite number of steps. We will see in later sections how the negation as failure–approach can be adapted in order to handle PROLOG(+T)–programs.

5.1.4. Programs

As in PROLOG, a program in PROLOG(+T) is given as a finite set of PROLOG(+T)–rules (and facts). The variables in the rules will be assumed to be *universally quantified.* This is a useful assumption as we will see when studying the semantics of PROLOG(+T)–programs in chapter 5.2.

Definition 5.1.5 (PROLOG(+T)–program)
A PROLOG(+T)*–program* is a finite set P of PROLOG(+T)–rules.

PROLOG(+T)–programs (or simply *programs* if there's no way of confusion) will be denoted as $P, \bar{P}, P' \dots$ from now on. Due to the special form of the formulas in programs it is possible to build a formula which is semantically equivalent to a PROLOG(+T)–program and which has a very simple form. Let P be a program containing n rules. That is let $P = \{P_1, \dots, P_n\}$ where each P_i has the form

$$P_i = \psi^{(i)} :\!- \varphi_1^{(i)}, \dots, \varphi_{n_i}^{(i)}.$$

for $\psi^{(i)} \in \mathcal{A}_t(\text{sig})$ and $\varphi_1^{(i)}, \dots, \varphi_{n_i}^{(i)} \in \mathcal{L}_t(\text{sig})$ and some $n_i \in \mathbb{N}$. For formal reasons we will identify $\psi^{(i)} :\!-$ and $\psi^{(i)}$. Due to the interpretation of clauses as sets of literals we therefore have

$$\begin{aligned} P_i &\equiv \neg\left(\varphi_1^{(i)} \wedge \dots \wedge \varphi_{n_i}^{(i)}\right) \vee \psi^{(i)} \\ &\equiv \neg\varphi_1^{(i)} \vee \dots \vee \neg\varphi_{n_i}^{(i)} \vee \psi^{(i)} \\ &\equiv \psi^{(i)} \vee \bigvee_{j=1}^{n_i} \neg\varphi_j^{(i)} \end{aligned}$$

for $i = 1, \dots, n$. Since sets of clauses are satisfied under interpretations if all clauses are satisfied simultaneously we have $\mathcal{J} \models P$ if and only if $\mathcal{J} \models P_i$ for $i = 1, \dots, n$ if and only if $\mathcal{J} \models P_1 \wedge \cdots \wedge P_n$ if and only if $\mathcal{J} \models \bigwedge_{i=1}^n P_i$. This gives $P \equiv \bigwedge_{i=1}^n P_i$ and therefore

$$\begin{aligned} P &\equiv \bigwedge_{i=1}^{n} P_i \\ &= \bigwedge_{i=1}^{n} \left(\psi^{(i)} \vee \bigvee_{j=1}^{n_i} \neg\varphi_j^{(i)} \right) \end{aligned}$$

is a relatively simple formula which might characterize the semantics of a PROLOG(+T)–program P.

From now on we will assume that the rules in a program P are *standardized apart.*

Definition 5.1.6

Let φ_1 and φ_2 be FOLTL–formulas. φ_1 and φ_2 are called *standardized apart* if $\textsc{Var}(\varphi_1) \cap \textsc{Var}(\varphi_2) = \emptyset$.

It is obvious that two rules from a program can be easily standardized apart. This is a consequence of the assumption that all variables in rules are implicitly universally quantified: let P be a program as described above and let $i, j \in \{1, \ldots, n\}$ be such that $\textsc{Var}(P_1) \cap \textsc{Var}(P_2) = \{\mathtt{X}_{i_1}, \ldots, \mathtt{X}_{i_k}\} \neq \emptyset$. We now fix two sets $\left\{\mathtt{X}_{i_1}^{(1)}, \ldots, \mathtt{X}_{i_k}^{(1)}\right\}$ and $\left\{\mathtt{X}_{i_1}^{(2)}, \ldots, \mathtt{X}_{i_k}^{(2)}\right\}$ such that $\left\{\mathtt{X}_{i_1}^{(1)}, \ldots, \mathtt{X}_{i_k}^{(1)}, \mathtt{X}_{i_1}^{(2)}, \ldots, \mathtt{X}_{i_k}^{(2)}\right\} \cap (\textsc{Var}(P_1) \cup \textsc{Var}(P_2)) = \emptyset$, define substitutions

$$\begin{aligned} \sigma_i &= \left\{ \frac{\mathtt{X}_{i_1}}{\mathtt{X}_{i_1}^{(i)}}, \ldots, \frac{\mathtt{X}_{i_k}}{\mathtt{X}_{i_k}^{(i)}} \right\} \text{ and} \\ \sigma_j &= \left\{ \frac{\mathtt{X}_{i_1}}{\mathtt{X}_{i_1}^{(j)}}, \ldots, \frac{\mathtt{X}_{i_k}}{\mathtt{X}_{i_k}^{(j)}} \right\} \end{aligned}$$

and replace the program P by $\bar{P} = (P \setminus \{P_i, P_j\}) \cup \{\sigma_i(P_i), \sigma_j(P_j)\}$. Due to the implicit universal quantification of all variables in the rules of P we have $P \models \varphi$ if and only if $\bar{P} \models \varphi$ for any φ.

5.1.5. Queries

As in PROLOG, a PROLOG(T)–program is *run* by giving a *query* to the program and searching for a substitution such that the result of applying this substitution to the query yields a logical consequence of the program. Formally we will adopt the philosophy of logic programming as described in chapter 3.1. A *query* in PROLOG(+T) is therefore interpreted as a sequence of single queries which have to be solved one by one. As in rules we will also allow negation as failure in queries.

Definition 5.1.7 (PROLOG(+T)–query)
Let sig $= (\mathcal{X}, F, P, \alpha)$ be a signature. A PROLOG(+T)–*query* over sig is a formula of the form

$$G = \varphi_1, \ldots, \varphi_n.,$$

such that $\varphi_1, \ldots, \varphi_n \in \mathcal{L}_t(\text{sig})$.

As we are interested in substitutions for the variables in a goal, we need the concept of an *answer* as described by [105]. An *answer* for a query G is a substitution σ such that $\text{DOM}(\sigma) = \text{VAR}(G)$. If P is a PROLOG(+T)–program and $G = \varphi_1, \ldots, \varphi_n.$, then an answer σ is called *correct* for G if $P \models \sigma(\varphi_1) \wedge \cdots \wedge \sigma(\varphi_n)$. Obviously the semantics of a program P can be characterized in terms of the set of all goals G consisting of a single query (i.e. $G = \varphi.$ for some $\varphi \in \mathcal{L}_t(\text{sig})$) such that there is a correct answer σ for G. In this case $\sigma(\varphi)$ is contained in the set characterizing the semantics of the program. How these characterization can be formally described will be the subject of the following section of this chapter.

5.1.6. The Relation of Prolog(+T) to other Temporal Logic programming Languages

This section will deal with the question of how PROLOG(+T) differs from other temporal logic programming languages introduced so far. In particular we will discuss the differences between PROLOG(+T), TEMPLOG and TOKIO. It will turn out that on the one hand PROLOG(+T) is syntactically closely related to TEMPLOG although there are some differences which are worth pointing out. On the other hand we will discuss TOKIO which is also similar to PROLOG(+T) regarding the temporal operators used in its definition but which has a completely different philosophical origin.

Templog

TEMPLOG is a first–order temporal programming language which has originally been introduced and defined by Abadi and Manna in [3] and [2]. The underlying logic of TEMPLOG allows the usage of the same set of temporal operators as PROLOG(+T) while the authors distinguish between *flexible* and *rigid* symbols. Flexible symbols may be interpreted as operations with a semantics which changes over time while the interpretations of rigid symbols must not be depending on the point of time at which the symbol is evaluated.

In [3] the definition of TEMPLOG is carried out as follows:

1. The authors define a temporal logic on which TEMPLOG is based.
2. They define a fragment of TEMPLOG in order to present the basic ideas.
3. Finally they introduce the full logic by enhancing the set of temporal operators which are allowed in the definition of the programming statements of the programming language TEMPLOG.

The fragment of TEMPLOG which is defined in [3] introduces the concepts of *initial temporal hornclauses* and *permanent temporal hornclauses* as an extension of the hornclauses which are used as programming statements in PROLOG. An initial temporal hornclause is a construct which has the form

$$\forall x_1 \dots \forall x_k \, (\varphi_1 \wedge \dots \wedge \varphi_n \to \psi)$$

for so called *next–atomic* formulas $\varphi_1, \dots, \varphi_n$ and ψ where $\bigcup_{i=1}^{n} \textsc{Var}(\varphi_i) \cup \textsc{Var}(\psi) = \{x_1, \dots, x_n\}$. Here a formula φ is called *next–atomic* if φ has the form $\varphi = \mathsf{X}^i \bar{\varphi}$ for some $i \geq 0$ and some atom $\bar{\varphi}$. Due to the fact that the definition TEMPLOG is motivated by the wish to enhance PROLOG with temporal operators the authors of [3] introduce the abbreviation

$$\psi \leftarrow \varphi_1, \dots, \varphi_n := \forall x_1 \dots \forall x_k \, (\varphi_1 \wedge \dots \wedge \varphi_n \to \psi)$$

which clearly shows the relation between TEMPLOG and PROLOG.

Additionally *permanent temporal hornclauses* are programming statements of the form

$$\forall x_1 \dots \forall x_k \mathsf{G} \, (\varphi_1 \wedge \dots \wedge \varphi_n \to \psi)$$

for next–atomic formulas $\varphi_1, \dots, \varphi_n$ and ψ and $\{x_1, \dots, x_k\}$ as before. Similar to initial temporal hornclauses permanent temporal hornclauses may be abbreviated using

$$\psi \Leftarrow \varphi_1, \dots, \varphi_n := \forall x_1 \dots \forall x_k \mathsf{G} \, (\varphi_1 \wedge \dots \wedge \varphi_n \to \psi) \, .$$

Programs in the fragment defined in this way are defined as sets consisting of permanent and initial hornclauses. Queries to such programs are defined as conjunctions of next–atomic formulas. It is possible to show that the programming language defined by this fragment can be evaluated using a resolution–style theorem proving procedure (see e.g. [1] for a discussion of such a procedure).

After having defined the fragment presented above the authors introduce the full logic programming TEMPLOG by allowing usage of G in the head and F in the tail of program statements (both with restrictions).

As a summary we can see:

1. TEMPLOG offers a distinction between flexible and rigid symbols which allows to interpret symbols in a different way at different points of time.

2. PROLOG(+T) does not have a counterpart to the operator $\Leftarrow$ of TEMPLOG.

3. In contrast to PROLOG(+T) TEMPLOG limits the application of G and F to the head (resp. the tail) of clauses.

4. The operators U and P are forbidden in TEMPLOG programs.

Tokio

TOKIO is another programming language which allows the usage of certain temporal operators in programming statements. TOKIO has been presented in [6] as an extention of the logic ITL (see e.g. [117]). In contrast to the logic underlying both TEMPLOG and PROLOG(+T) TOKIO and its predecessor ITL are *interval based* logic programming languages. This means that the main goal of a proof procedure executing a TOKIO program is not to prove a goal but to find an interval of time in which the goal holds. We will make this clear soon.

The syntax of TOKIO's programming constructs is defined in [6] (in another paper an interpreter for TOKIO written in PROLOG is presented. It is noted that the execution time ofTOKIO programs using this interpreter is slowed down by a factor of 40 in contrast to *ordinary* PROLOG programs; we refer to [92] for implementation details). Similarly to PROLOG we can use programming statements which do not contain temporal operators. So every PROLOG–statement $\psi \leftarrow \varphi_1, \ldots, \varphi_n$ is also a well–formed TOKIO–statement.

Additionally to PROLOG TOKIO allows the usage of several temporal constructs. These are

- sequential execution,

- the Next–operator,
- the Always–operator,
- the Sometimes–operator,
- the *keep*–operator and
- the *fin*–operator.

Since TOKIO is an interval–based programming approach the main goal of a prover executing a TOKIO formula is to define an interval of time in which this formula is satisfied. Time in TOKIO is considered discrete, so an interval I can be described by specifying its start– resp. endpoint in terms of natural numbers. If $I = [I_{\text{beg}}, I_{\text{fin}}]$ is such an interval we have to require that $I_{\text{beg}} \leq I_{\text{fin}}$. Since the motivation for the definition of TOKIO is the description of hardware the authors of [6] prefer the term *execution* in order to denote the satisfaction of a formula.

We will now make the concepts of TOKIO more clear.

1. TOKIO allows the specification of the sequential execution of goals. This is carried out by the so called *chop*–operator which is denoted by `&&`. So a statement $\psi \leftarrow \varphi_1 \texttt{\&\&} \ldots \texttt{\&\&} \varphi_n$ is executed in an interval $I = [I_{\text{beg}}, I_{\text{fin}}]$ if I can be divided into intervals $I_1, \ldots, I_n$ such that $I_1 = [I_{\text{beg}}, t_1], \ldots, I_j = [t_{j-1}, t_j], \ldots, I_n = [t_{n-1}, I_{\text{fin}}]$ for $t_1, \ldots, t_{n-1} \in \mathbb{N}$ such that $t_i \leq t_{i+1}$ for $i = 1, \ldots, n-2$ and each φ_i is executed in the interval I_i.

2. The Next–operator is intended as a similar concept to the operator X from PROLOG(+T). In TOKIO the Next–operator is denoted as `@` and a statement $\psi \leftarrow \texttt{@}\varphi$ is intended to be executed in an interval $I = [I_{\text{beg}}, I_{\text{fin}}]$ if ψ is executed in the interval $[I_{\text{beg}} + 1, I_{\text{fin}}]$.

3. Similarly the Always– and Sometimes–operators which are denoted as `#` resp. `<>`

require the execution of a statement at certain points of time in an interval. If $I = [I_{\text{beg}}, I_{\text{fin}}]$ is a given interval, then

a) $\psi \leftarrow \texttt{\#}\varphi$ is executed in I if φ is executed at every point $I_{\text{beg}}, \ldots, I_{\text{beg}}+j, \ldots, I_{\text{fin}}$ of time and

b) $\psi \leftarrow \texttt{<>}\varphi$ is executed in I if there is (at least) one j with $0 \leq j \leq \text{fin} - \text{beg}$ such that φ is executed at $I_{\text{beg}} + j$.

4. The *keep*–operator allows reasoning about the execution of a statement at serval points of time in an interval. If I is given then $\psi \leftarrow \mathit{keep}(\varphi)$ is executed in I if φ is executed at every point of time in I except of the last point. More formally if $I = [I_{\text{beg}}, I_{\text{fin}}]$ then $\psi \leftarrow \mathit{keep}(\varphi)$ is executed in I if φ is executed at time $I_{\text{beg}}, \ldots, I_{\text{fin}} - 1$ and φ is *not* executed at time point I_{fin}.

5. Finally the *fin*–operator is introduced in order to reason about the execution of a statement at a final point of time in an interval. If $I = [I_{\text{beg}}, I_{\text{fin}}]$ is given then $\psi \leftarrow \mathit{fin}(\varphi)$ is executed in I if φ is executed at the time I_{fin}.

Although the operators `#`, `&&` and `@` are the same operators as their counterparts in PROLOG(+T) the *keep* and *fin*–operators cannot be mapped to PROLOG(+T) adequately. The only relationships we can derive is that

1. statements of the form $\psi \leftarrow \mathit{keep}(\varphi)$ are related to statements of the form $\psi :\!- \varphi \mathsf{U} \bar{\varphi}$ for *some* $\bar{\varphi}$ and

2. statements of the form $\psi \leftarrow \mathit{fin}(\varphi)$ are related to statements of the form $\psi :\!- \varphi \mathsf{P} \bar{\varphi}$ for *some* $\bar{\varphi}$.

As a summary we can therefore point out the following differences between TOKIO and PROLOG(+T):

1. PROLOG(+T) does not have direct counterparts for the operators *keep*, *fin* and `&&` of TOKIO.

2. In PROLOG(+T) we do not have any limits for the time points in which formulas are satisfied. In contrast TOKIO statements are executed in intervals. Since such intervals are assumed to be given by time points which are represented as natural numbers, they always have a finite length.

3. The philosophies of TOKIO and PROLOG(+T) are completely different since the main task in TOKIO is the construction of a model (i.e. the detection of an interval) while the main task in PROLOG(+T) is to refute a formula, i.e. proving that there cannot exist any models.

5.2. Declarative Semantics of Prolog(+T)

This section will deal with the *declarative semantics* of the programming language PROLOG(+T) introduced in the last section. The declarative semantics is defined in terms of the logical consequence relation $\models$ and it is important to distinguish this semantics from the *operational semantics* which is defined in terms of some suitable calculus $\vdash$. This operational semantics will be the subject of chapter 5.3.

5.2.1. Preliminaries

We will now extend the concepts of *Herbrand–interpretations* as introduced in first order logic (see [140] or [105]) to the logic FOLTL. We will see that the results from first order logic carry over to FOLTL.

Let sig $= (\mathcal{X}, F, P, \alpha)$ be any signature. We will need the concept of *ground objects* in order to define Herbrand–interpretations.

- A term $t \in \mathcal{T}(\text{sig})$ is called a *ground term* if $\text{VAR}(t) = \emptyset$.

- A FOLTL–formula φ is called a *ground formula* if $\text{VAR}(\varphi) = \emptyset$.

Similarly a *ground atom* is a ground formula which is an atom, a *ground literal* is a ground formula which is a literal and a *ground clause* is a ground formula which is a clause.

For the rest of this chapter we will assume that $\mathrm{sig} = (\mathcal{X}, F, P, \alpha)$ is any fixed signature. Let Φ be any set of formulas over sig. The set of all ground terms which can be built from symbols occurring in Φ will be called the *Herbrand–universe* of Φ and will be denoted as U_Φ. In particular we will be interested in U_P for Prolog(+T)–programs P. Similarly one defines sets B_Φ and $B_\Phi^{\textsc{FoLtl}}$ as follows:

$$\begin{aligned} B_\Phi &= \{\varphi \in \mathcal{A}(\mathrm{sig}) \mid \textsc{Var}(\varphi) = \emptyset\} \text{ and} \\ B_\Phi^{\textsc{FoLtl}} &= \{\varphi \in \mathcal{A}_t(\mathrm{sig}) \mid \textsc{Var}(\varphi) = \emptyset\} \end{aligned}$$

The set B_Φ is the well known *Herbrand–base*. The set $B_\Phi^{\textsc{FoLtl}}$ is an extension of the Herbrand–base which also allows the inclusion of temporal atoms. We will also refer to $B_\Phi^{\textsc{FoLtl}}$ as the *Herbrand–base* of Φ since there is no way of confusion[2].

We will see that a well-known result from first order logic can be extended to FoLtl. Therefore we will define the set $\textsc{Free}(o)$ for some logical object o to be the set of *free variables* occurring in this object. Formally:

- for terms from $\mathcal{T}(\mathrm{sig})$ we define
  - $\textsc{Free}(\mathtt{X}) = \{\mathtt{X}\}$ if $\mathtt{X} \in \mathcal{X}$ is a variable symbol and
  - $\textsc{Free}(\mathtt{f}(t_1, \ldots, t_n))) = \bigcup_{i=1}^n \textsc{Free}(t_i)$ if $\mathtt{f} \in F$ is a function symbol with $\alpha(\mathtt{f}) = n$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathrm{sig})$.

  Equivalently we could define $\textsc{Free}(t) = \textsc{Var}(t)$ for any $t \in \mathcal{T}(\mathrm{sig})$.

- For formulas we define

[2]Due to the fact that syntactically different literals might be logically equivalent we have that a *positive* literal can be equivalent to a *negative* one, e.g. $\mathsf{G}p(\mathtt{X}) \equiv \mathtt{not}(\mathsf{F}\mathtt{not}(p(\mathtt{X})))$. Therefore we will make the following convention: $B_P^{\textsc{FoLtl}}$ contains all temporal literals which are equivalent to some positive temporal literal.

– $\textsc{Free}(\texttt{true}) = \textsc{Free}(\texttt{false}) = \emptyset$,

– if $\texttt{p} \in P$ is a predicate symbol with $\alpha(\texttt{p}) = n$ and $t_1, \ldots, t_n \in \mathcal{T}(\text{sig})$, then $\textsc{Free}(\texttt{p}(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \textsc{Free}(t_i)$,

– if φ is a formula, then

$$\textsc{Free}(\texttt{not}(\varphi)) = \textsc{Free}(\mathsf{X}\varphi) = \textsc{Free}(\mathsf{G}\varphi) = \textsc{Free}(\mathsf{F}\varphi) = \textsc{Free}(\varphi),$$

– if φ is formula, then $\textsc{Free}(\forall \texttt{X}\varphi) = \textsc{Free}(\exists \texttt{X}\varphi) = \textsc{Free}(\varphi) \setminus \{\texttt{X}\}$ and

– if φ_1 and φ_2 are formulas, then $\textsc{Free}(\varphi_1 \mathsf{U} \varphi_2) = \textsc{Free}(\varphi_1 \mathsf{P} \varphi_2) = \textsc{Free}(\varphi_1 \wedge \varphi_2) = \textsc{Free}(\varphi_1 \vee \varphi_2) = \textsc{Free}(\varphi_1 \rightarrow \varphi_2) = \textsc{Free}(\varphi_1 \leftrightarrow \varphi_2) = \textsc{Free}(\varphi_1) \cup \textsc{Free}(\varphi_2)$.

A formula φ is called *closed* if $\textsc{Free}(\varphi) = \emptyset$. Similarly a formula φ is called *universally closed* if φ is a closed formula which does not contain the quantifier $\exists$.

We will now see that certain subsets of the Herbrand–base of a program can be considered as interpretations. This is achieved in a similar way as in the case of first order logic programs. However, we will have to put some restrictions on the subsets of interest. After this we will make a link between these sets of atoms (which we will refer to as *set–based–interpretations*) and the interpretations of FOLTL–formulas as defined in chapter 2.3.2 (which we will refer to as *structure–based interpretations*).

Let $P = \{P_1, \ldots, P_n\}$ be any PROLOG(+T)–program and let $I \subseteq B_P^{\textsc{FoLtl}}$ be any set of ground atoms built from symbols occurring in P. Furthermore let φ be any universally closed FOLTL–formula[3]. I will be seen as an interpretation for φ as follows:

1. if $\varphi = \mathsf{X}^i\psi$ for any $i \geq 0$ and any nontemporal atom ψ from B_P, then $I \models \varphi$ if and only if $\varphi \in I$,

[3]Similarly as in First Order Logic these construction strongly relies on the assumption that every variable symbol in the formula under consideration is universally quantified. For general formulas the construction fails.

2. if $\varphi = \varphi_1 \wedge \varphi_2$, then $I \models \varphi$ if and only if $I \models \varphi_1$ and $I \models \varphi_2$,

3. if $\varphi = \varphi_1 \vee \varphi_2$, then $I \models \varphi$ if and only if $I \models \varphi_1$ or $I \models \varphi_2$,

4. if $\varphi = \neg\psi$, then $I \models \varphi$ if and only $I \not\models \psi$,

5. if $\varphi = \varphi_1 \rightarrow \varphi_2$, then $I \models \varphi$ if and only if $I \models \neg\varphi_1$ or $I \models \varphi_2$,

6. if $\varphi = \varphi_1 \leftrightarrow \varphi_2$, then $I \models \varphi$ if and only if $I \models \varphi_1 \rightarrow \varphi_2$ and $I \models \varphi_2 \rightarrow \varphi_1$ and

7. if $\varphi = \forall \mathtt{X}\psi$ then

   a) if $\mathtt{X} \in \mathrm{VAR}(\psi)$, then $I \models \varphi$ if and only if for every substitution $\sigma = \left\{\frac{\mathtt{X}}{t}\right\}$ for some t such that $\mathrm{VAR}(t) = \emptyset$ it holds that $I \models \sigma(\psi)$ and

   b) if $\mathtt{X} \notin \mathrm{VAR}(\psi)$, then $I \models \varphi$ if and only if $I \models \psi$.

This definition of the semantics in terms of subsets of B_P^{FoLtl} only allows the interpretation of formulas which contain the temporal operator X. But in order to handle formulas involving G, F, U and P we have to restrict the subsets of interest to such subsets which are *temporally closed*.

Definition 5.2.1 (Temporally closed set)

A set $I \subseteq B_P^{\mathrm{FoLtl}}$ is called *temporally closed* if and only if for every φ, φ_1 and φ_2 from B_P^{FoLtl} and every $i \geq 0$ the following conditions are fulfilled:

1. $\mathsf{X}^i\mathsf{G}\varphi \in I$ if and only if $\mathsf{X}^{i+j}\varphi \in I$ for every $j \geq 0$,

2. $\mathsf{X}^i\mathsf{F}\varphi \in I$ if and only if $\mathsf{X}^{i+j}\varphi \in I$ for some $j \geq 0$,

3. $\mathsf{X}^i\varphi_1\mathsf{U}\varphi_2 \in I$ if and only if $\mathsf{X}^i\varphi_2 \in I$ or $\mathsf{X}^i\varphi_1 \in I$ and $\mathsf{X}^{i+1}\varphi_1\mathsf{U}\varphi_2 \in I$,

4. $\mathsf{X}^i\varphi_1\mathsf{P}\varphi_2 \in I$ if and only if $\mathsf{X}^i\varphi_2 \notin I$ and $\mathsf{X}^i\varphi_1 \in I$ or $\mathsf{X}^{i+1}\varphi_1\mathsf{P}\varphi_2 \in I$ and

5. $\varphi \in I$ if and only if $\left\{\psi \in B_P^{\mathrm{FoLtl}} \mid \psi \equiv \varphi\right\} \subseteq I$.

The motivation for this definition should immediately be clear since it is directly derived from the properties of the semantical equivalence relation for FOLTL–formulas. So a set I is temporally closed if satisfaction of *one* member of an equivalence class implies satisfaction of *all* members of this equivalence class. Therefore considering temporally closed subsets of B_P^{FoLtl} as the interpretations of interest is reasonable.

It is important to note that one can in many cases enrich a set $I \subseteq B_P^{\text{FoLtl}}$ by adding atoms in order to receive a temporally closed set. This procedure will later be referred to as *building* (or *constructing*) *the temporal closure.* In general, a temporally closed superset of a set I is not uniquely determined.

Example 5.2.1

Consider the set $I = \{\mathsf{Fp(a)}\}$. every temporally closed superset of I contains $\mathsf{X}^j p(a)$ for some $j \geq 0$.

The temporally closed supersets of a set I can be seen as the unions of sets on (infinite) maximal paths in an infinite tree. We will therefore construct a labeled graph $T(I) = (V, E, l)$ from I as described below[4].

$T(I) = (V, E, l)$ with $l : V \to 2^{B_P^{\text{FoLtl}}} \cup \{\texttt{fail}\}$ is the infinite tree satisfying the following conditions:

1. There is a uniquely determined $v_0 \in V$ such that $(v, v_0) \notin E$ for each $v \in V$ (the *root* node),

2. $l(v_0) = I$ and

3. for each $v \in V$ the following is true:

   a) if there is an atom $\mathsf{X}^i\mathsf{G}\varphi \in l(v)$, then there is $v' \in V$ such that $(v, v') \in E$ and $l(v') = l(v) \cup \{\mathsf{X}^{i+j}\varphi \mid j \geq 0\}$,

[4]Here V is a nonempty set of *nodes* (also called *vertices*), $E \subseteq V \times V$ is a set of *edges* connecting these nodes and $l : v \to 2^{B_P^{\text{FoLtl}}} \cup \{\texttt{fail}\}$ is a mapping which *labels* the nodes.

b) if there is an atom $\mathsf{X}^i\mathsf{F}\varphi \in l(v)$ then there are $v'_1, \ldots, v'_j, \cdots \in V$ such that $(v, v'_j) \in E$ and $l(v'_j) = l(v) \cup \{\mathsf{X}^{i+j}\varphi\}$ for each $j \geq 0$,

c) if there is an atom $\mathsf{X}^i\varphi_1\mathsf{U}\varphi_2 \in l(v)$, then there are $v'_1, v'_2 \in V$ such that $(v, v'_1), (v, v'_2) \in E$ and

- $l(v'_1) = l(v) \cup \{\mathsf{X}^i\varphi_2\}$ and
- $l(v'_2) = l(v) \cup \{\mathsf{X}^{i+1}\varphi_1\mathsf{U}\varphi_2\}$

if $\mathsf{X}^i\varphi_2 \notin l(v)$,

d) if there is an atom $\mathsf{X}^i\varphi_1\mathsf{P}\varphi_2 \in l(v)$ then there are $v'_1, v'_2 \in V$ such that $(v, v'_1), (v, v'_2) \in E$ and

- if $\mathsf{X}^i\varphi_2 \in l(v)$, then $l(v'_1) = l(v'_2) = \texttt{fail}$ and
- if $\mathsf{X}^i\varphi_2 \notin l(v)$, then $l(v'_1) = l(v) \cup \{\mathsf{X}^i\varphi_1)\}$ and $l(v'_2) = l(v) \cup \{\mathsf{X}^{i+1}\varphi_1\mathsf{P}\varphi_2\}$.

and

e) for each $\varphi \in l(v)$ it holds that

$$\{\psi \in B_P^{\text{FoLtl}} \mid \psi \equiv \varphi\} \subseteq l(v).$$

Given $T(I)$ the set of all (possibly infinite) maximal paths π starting at v_0 such that there is no v with $l(\pi) = \texttt{fail}$ occurring on π will be denoted as $p(T(I))$.

Given $\pi \in p(T(I))$, the set $l(\pi)$ denotes the union of all sets with which the nodes on π are labeled. Formally if $V(\pi)$ denotes the set of nodes occurring on π, then

$$l(\pi) = \bigcup_{v \in V(\pi)} l(v).$$

Then the following claims are immediate:

1. For every $\pi \in p(T(I))$ the set $l(\pi)$ is temporally closed.

2. For every $\pi \in p(T(I))$ it holds that $I \subseteq l(\pi)$.

We will from now on denote the set consisting of all sets computable in the way sketched above as the *temporal closure* of I and denote it as

$$\textsc{TempClosure}(I) = \{l(\pi) \mid \pi \in p(T(I))\} .$$

Definition 5.2.2 (Set–based Herbrand–Interpretation)
A *set–based Herbrand–Interpretation* for a program P is a temporally closed subset $I \subseteq B_P^{\textsc{FoLtl}}$. A set–based *Herbrand–model* for P is any temporally closed set $I \subseteq B_P^{\textsc{FoLtl}}$ such that $I \models P$.

It is worth noticing that considering only temporally closed sets as interpretations has the drawback that there are some programs which are not satisfiable. This is one difference to pure first order logic programs which are always satisfied by the interpretation $I = B_P$. Consider the following program $P = \{P_1, P_2\}$ where

$$\begin{aligned} P_1 &= \mathtt{p(a)}.\text{ and} \\ P_2 &= \mathtt{q(X)Pp(X)} \text{ :--} \mathtt{p(X)}. \end{aligned}$$

Now fix any temporally closed $I \subseteq B_P^{\textsc{FoLtl}}$. If $I \models P$, then in particular we have $I \models P_1$, that is $I \models P_1 = \mathtt{p(a)}$. So $\mathtt{p(a)} \in I$. On the other hand we have $I \models P_2 = \mathtt{q(X)Pp(X)}$: $-\mathtt{p(X)}$ and since P_2 is considered universally closed we have $I \models \mathtt{q(a)Pp(a)}$:–$\mathtt{p(a)}$. But since I is temporally closed we have $I \not\models \mathtt{p(a)}$ and $I \models \mathtt{q(a)}$ or $I \models \mathtt{Xq(a)Pp(a)}$. This is a contradiction. So $I \not\models P$ and therefore P has no set–based Herbrand–model.

We will now prove that for universally closed sets of FOLTL–formulas the concepts of set–based Herbrand–Interpretations and structure–based Herbrand–Interpretations as introduced in chapter 2.3.2 are equivalent. This allows reasoning about properties of programs by considering interpretations as sets of literals instead of the formally more complicated structures.

We will first prove the *easier* direction.

Lemma 5.2.1

Let P be a PROLOG(+T)–program. If P has a structure–based Herbrand–model, then P has a set–based Herbrand–model.

Proof. Let P be a PROLOG(+T)–program and let $\mathcal{J} = (U_{\mathcal{J}}, S, s_0, \delta_1, \delta_2, w, \mathcal{I})$ be a structure–based Herbrand–model of P. Set $I_{\mathcal{J}} = \{\varphi \in B_P^{\text{FoLtl}} \mid \mathcal{J} \models \varphi\}$, i.e. $I_{\mathcal{J}}$ becomes the set of all ground instances satisfied by $\mathcal{J}$. Then every $I \in \text{TempClosure}(I_{\mathcal{J}})$ is easily seen to be a set–based Herbrand–model of P. □

The opposite direction is also true although it is much more complicated to prove (due to the more complicated definition of structure–based interpretations).

Lemma 5.2.2

Let P be a PROLOG(+T)–program. If P has a set–based Herbrand–model, then P has a structure–based Herbrand–model.

Proof. Let $I \subseteq B_P^{\text{FoLtl}}$ be a set–based Herbrand–model of P, that is I is temporally closed and $I \models P$. We will construct a structure–based interpretation $\mathcal{J}_I = (U_P, S, s_0, \delta_1, \delta_2, w, \mathcal{I})$ from I such that $\mathcal{J}_I \models P$. Therefore define $S = \{s^{(i)} \mid i \in \mathbb{N}\}$ and $s_0 = s^{(0)}$. Furthermore define for $s \in S$ and any $t = f(t_1, \ldots, t_n) \in U_P$: $\mathcal{I}(s,t) = \mathcal{I}(s,f)(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ as obvious. Since every element of P is considered universally closed we can set w as any arbitrary mapping.

We then define $\delta_1 := \{(s^{(i)}, s^{(i+1)}) \mid i \geq 0\}$ and $\delta_2 := \{(s^{(i)}, s^{(i+j)}) \mid i, j \geq 0\}$. After this we proceed as follows:

1. Take some φ from I and set $I = I \setminus \{\varphi\}$.

2. **Case 1** if $\varphi = \mathsf{X}^i \mathtt{p}(t_1, \ldots, t_n)$ for some $i \geq 0$, some predicate symbol $\mathtt{p}$ of arity n

and $t_1, \ldots, t_n \in U_P$, then we set

$$\mathcal{I}\left(s^{(i)}, \mathtt{p}\right)(t_1, \ldots, t_n) = 1,$$

Case 2 if $\varphi = \mathsf{X}^i\mathsf{G}\psi$ for some $i \geq 0$ and some $\psi \in B_P^{\text{FoLtl}}$, then set

$$\mathcal{I}\left(s^{(i+j)}, \psi\right) = 1$$

for every $j \geq 0$,

Case 3 if $\varphi = \mathsf{X}^i\mathsf{F}\psi$ for some $i \geq 0$ and some $\psi \in B_P^{\text{FoLtl}}$, then set

$$\mathcal{I}\left(s^{(i+j)}, \psi\right) = 1$$

for some $j \geq 0$,

Case 4 if $\varphi = \mathsf{X}^i\varphi_1\mathsf{U}\varphi_2$ for some $i \geq 0$ and $\varphi_1, \varphi_2 \in B_P^{\text{FoLtl}}$, then set

$$\mathcal{I}\left(s^{(i)}, \varphi_2\right) = 1$$

or

$$\mathcal{I}\left(s^{(i)}, \varphi_1\right) = 1 \text{ and } \mathcal{I}\left(s^{(i+1)}, \varphi_1\mathsf{U}\varphi_2\right) = 1$$

and

Case 5 if $\varphi = \mathsf{X}^i\varphi_1\mathsf{P}\varphi_2$ for some $i \geq 0$ and $\varphi_1, \varphi_2 \in B_P^{\text{FoLtl}}$, then set

$$\mathcal{I}\left(s^{(i)}, \varphi_2\right) = 0$$

and

$$\mathcal{I}\left(s^{(i)}, \varphi_1\right) = 1 \text{ or } \mathcal{I}\left(s^{(i+1)}, \varphi_1\mathsf{P}\varphi_2\right) = 1.$$

3. If $I \neq \emptyset$, then go back to step 1.

It is obvious that the structure–based interpretation $\mathcal{J}_I$ which emerges by applying these steps as long as $I \neq \emptyset$ (i.e. the *limit interpretation*) is a structure–based Herbrand–model of P. $\Box$

Theorem 5.2.1 (Equivalence of set– and structure–based interpretations) Let P be any PROLOG(+T)–program. Then P has a structure–based Herbrand–model if and only if P has a set–based Herbrand–model.

Proof. Immediately from Lemma 5.2.1 and Lemma 5.2.2. $\Box$

5.2.2. Reduction of Literals

It will turn out useful to introduce a concept of *reduction* for temporal literals. Intuitively a reduced form of some temporal literal φ is a *normal form* $\textsc{Red}(\varphi)$. We will therefore define how a reduced literal looks like and how it can be effectively computed given the (unreduced) literal.

An approach to define a certain type of normal–form for temporal logic formulas has been presented in [65] and [64] for the propositional logic LTL enriched with past operators. However, this *separated normal form* is defined for a much larger class of formulas than only atoms and literals. Consequently the structure of this normal form is much more complicated than necessary for our purposes.

In order to compute reduced forms of literals we will exploit several simple semantical equivalences. The basic idea is to first *pull out* the negation operator (if it is contained), so that each reduced literal φ is either of the form $\varphi = \psi$ for some literal ψ not containing `not` or $\varphi = \texttt{not}(\psi)$ for some ψ not containing `not`. The following set of logical equivalences will be the basis of our reduction concept.

$$
\begin{array}{rcllrcl}
\mathsf{GG}\varphi & \mapsto & \mathsf{G}\varphi, & & \mathsf{FF}\varphi & \mapsto & \mathsf{F}\varphi,\\
\mathsf{Gnot}(\varphi) & \mapsto & \mathtt{not}(\mathsf{F}\varphi), & & \mathsf{Fnot}(\varphi) & \mapsto & \mathtt{not}(\mathsf{G}\varphi),\\
\mathsf{Xnot}(\varphi) & \mapsto & \mathtt{not}(\mathsf{X}\varphi), & & \mathtt{not}(\mathtt{not}(\varphi)) & \mapsto & \varphi,\\
\mathsf{X}\mathtt{false} & \mapsto & \mathtt{false}, & & \mathsf{X}\mathtt{true} & \mapsto & \mathtt{true},\\
\mathsf{G}\mathtt{false} & \mapsto & \mathtt{false}, & & \mathsf{G}\mathtt{true} & \mapsto & \mathtt{true},\\
\mathsf{F}\mathtt{false} & \mapsto & \mathtt{false}, & & \mathsf{F}\mathtt{true} & \mapsto & \mathtt{true},\\
\mathtt{true}\mathsf{U}\varphi & \mapsto & \mathsf{F}\varphi, & & \mathtt{true}\mathsf{P}\varphi & \mapsto & \mathtt{not}(\varphi),\\
\varphi\mathsf{U}\mathtt{true} & \mapsto & \mathtt{true}, & & \varphi\mathsf{P}\mathtt{true} & \mapsto & \mathtt{false},\\
\mathtt{false}\mathsf{U}\varphi & \mapsto & \varphi, & & \mathtt{false}\mathsf{P}\varphi & \mapsto & \mathtt{not}(\mathsf{F}\varphi),\\
\varphi\mathsf{U}\mathtt{false} & \mapsto & \mathsf{G}\varphi & \text{and} & \varphi\mathsf{P}\mathtt{false} & \mapsto & \mathsf{F}\varphi.
\end{array}
$$

Figure 5.1.: Rewrite System for computing reduced literals

Lemma 5.2.3

Let φ be any formula from FOLTL. Then

$$
\begin{array}{rcllrcl}
\mathsf{GG}\varphi & \equiv & \mathsf{G}\varphi, & & \mathsf{FF}\varphi & \equiv & \mathsf{F}\varphi,\\
\mathsf{Gnot}(\varphi) & \equiv & \mathtt{not}(\mathsf{F}\varphi), & & \mathsf{Fnot}(\varphi) & \equiv & \mathtt{not}(\mathsf{G}\varphi),\\
\mathsf{Xnot}(\varphi) & \equiv & \mathtt{not}(\mathsf{X}\varphi), & & \mathtt{not}(\mathtt{not}(\varphi)) & \equiv & \varphi,\\
\mathsf{X}\mathtt{false} & \equiv & \mathtt{false}, & & \mathsf{X}\mathtt{true} & \equiv & \mathtt{true},\\
\mathsf{G}\mathtt{false} & \equiv & \mathtt{false}, & & \mathsf{G}\mathtt{true} & \equiv & \mathtt{true},\\
\mathsf{F}\mathtt{false} & \equiv & \mathtt{false}, & & \mathsf{F}\mathtt{true} & \equiv & \mathtt{true},\\
\mathtt{true}\mathsf{U}\varphi & \equiv & \mathsf{F}\varphi, & & \mathtt{true}\mathsf{P}\varphi & \equiv & \mathtt{not}(\varphi),\\
\varphi\mathsf{U}\mathtt{true} & \equiv & \mathtt{true}, & & \varphi\mathsf{P}\mathtt{true} & \equiv & \mathtt{false},\\
\mathtt{false}\mathsf{U}\varphi & \equiv & \varphi, & & \mathtt{false}\mathsf{P}\varphi & \equiv & \mathtt{not}(\mathsf{F}\varphi),\\
\varphi\mathsf{U}\mathtt{false} & \equiv & \mathsf{G}\varphi & \text{and} & \varphi\mathsf{P}\mathtt{false} & \equiv & \mathsf{F}\varphi.
\end{array}
$$

These equivalences are easily seen to be correct. In order to define a suitable concept of reducedness we will convert the equivalences into a terminating and confluent rewrite system. The rules of this system are then applied exhaustively to a literal and the resulting nonreducible literal is said to be the *reduced form* of the original literal. The set of rules is given in Figure 5.1.

In order to analyze the properties of the introduced rewrite system we will have to review some of the basic concepts from term rewriting. The presentation will be standard (see [13], [12] or [83]).

Let $t \in \mathcal{T}$ be a term. The set of *positions* of t is defined as

1. $\textsc{Pos}(\mathtt{X}) = \{\varepsilon\}$ if $t = \mathtt{X} \in \mathcal{X}$ and
2. $\textsc{Pos}(\mathtt{f}(t_1, \ldots, t_n) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \textsc{Pos}(t_i)\}$.

Similarly we define positions in literals as follows:

$$\begin{aligned}
\textsc{Pos}(\mathtt{true}) &= \textsc{Pos}(\mathtt{false}) = \{\varepsilon\},\\
\textsc{Pos}(\mathtt{p}(t_1, \ldots, t_n)) &= \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in \textsc{Pos}(t_i)\},\\
\textsc{Pos}(\neg\varphi) &= \textsc{Pos}(\varphi),\\
\textsc{Pos}(\mathsf{X}\varphi) &= \textsc{Pos}(\mathsf{G}\varphi) = \textsc{Pos}(\mathsf{F}\varphi)\\
&= \{\varepsilon\} \cup \{1p \mid p \in \textsc{Pos}(\varphi) \text{ and}\\
\textsc{Pos}(\varphi_1 \mathsf{U} \varphi_2) &= \textsc{Pos}(\varphi_1 \mathsf{P} \varphi_2)\\
&= \{\varepsilon\} \cup \{1p \mid p \in \textsc{Pos}(\varphi_1)\} \cup \{2p \mid p \in \textsc{Pos}(\varphi_2)\}.
\end{aligned}$$

So $\textsc{Pos}(o) \subseteq \mathbb{N}^*$ for any logical object o.

The term respectively literal at any given position can then be extracted as follows:

1. $t|_\varepsilon = t$ for each $t \in \mathcal{T}$ and
2. $\mathtt{f}(t_1, \ldots, t_i, \ldots, t_n)|_{ip} = t_i|_p$ for $p \in \textsc{Pos}(t_i)$.

Similarly we can extract subparts from literals:

1. $\varphi|_\varepsilon = \varphi$ for every literal φ,
2. $\mathtt{p}(t_1, \ldots, t_i, \ldots, t_n)|_{ip} = t_i|_p$ for $p \in \textsc{Pos}(t_i)$,
3. $\mathsf{X}\varphi|_{1p} = \mathsf{G}\varphi|_{1p} = \mathsf{F}\varphi|_{1p} = \varphi|_p$ for $p \in \textsc{Pos}(\varphi)$,

4. $(\varphi_1 \mathsf{U} \varphi_2)|_{1p} = (\varphi_1 \mathsf{P} \varphi_2)|_{1p} = \varphi_1|_p$ for $p \in \textsc{Pos}(\varphi_1)$ and

5. $(\varphi_1 \mathsf{U} \varphi_2)|_{2p} = (\varphi_1 \mathsf{P} \varphi_2)|_{2p} = \varphi_2|_p$ for $p \in \textsc{Pos}(\varphi_2)$.

Example 5.2.2

Let $t = \mathtt{f(f(g(X,f(a))))}$ and $\varphi = \mathtt{p(f(g(a,f(a))))}$ be given. Then

$$\begin{aligned}
\textsc{Pos}(t) &= \{\varepsilon, 1, 11, 111, 112, 1121\}\,, \\
\textsc{Pos}(\varphi) &= \textsc{Pos}(\mathtt{p(f(g(a,f(}a\mathtt{)))))}) \\
&= \{\varepsilon\} \cup \{1p \mid p \in \textsc{Pos}(\mathtt{f(g(a,f(a))))}\} \\
&= \{\varepsilon\} \cup \{1\varepsilon, 11, 111, 1111, 1112, 11121\} \\
&= \{\varepsilon, 1, 11, 111, 1111, 1112, 11121\}
\end{aligned}$$

and

$$\begin{aligned}
t|_{112} &= \mathtt{f(a)} & t|_{1121} &= \mathtt{a} \\
\varphi|_{11} &= \mathtt{f(g(a,f(a)))} & \varphi|_{11121} &= \mathtt{a}
\end{aligned}$$

The operation of *replacement* is now defined as follows (following the formalisms from [13]):

1. $t'[t]|_\varepsilon = t$ for $t, t' \in \mathcal{X}$,

2. $\mathtt{f}(t_1, \ldots, t_i, \ldots, t_n)[t]|_{ip} = \mathtt{f}(t_1, \ldots, t_{i-1}, t_i[t]|_p, t_{i+1}, \ldots, t_n)$ for $t \in \mathcal{X}$ and $p \in \textsc{Pos}(t_i)$,

3. $\varphi[\psi]|_\varepsilon = \neg\varphi[\psi]|_\varepsilon = \psi$ for literals φ, ψ,

4. a) $(\mathsf{X}\varphi[\psi])|_{1p} = \mathsf{X}(\varphi[\psi]|_p)$ for $p \in \textsc{Pos}(\varphi)$ if $\varphi|_p$ is a literal and

   b) $(\mathsf{X}\varphi[t])|_{1p} = \mathsf{X}(\varphi[t]|_p)$ for $p \in \textsc{Pos}(\varphi)$ if $\varphi|_p$ is a term,

5. a) $(\mathsf{G}\varphi[\psi])|_{1p} = \mathsf{G}(\varphi[\psi]|_p)$ for $p \in \textsc{Pos}(\varphi)$ if $\varphi|_p$ is a literal and

   b) $(\mathsf{G}\varphi[t])|_{1p} = \mathsf{G}(\varphi[t]|_p)$ for $p \in \textsc{Pos}(\varphi)$ if $\varphi|_p$ is a term,

6. a) $(\mathsf{F}\varphi[\psi])|_{1p} = \mathsf{F}(\varphi[\psi]|_p)$ for $p \in \textsc{Pos}(\varphi)$ if $\varphi|_p$ is a literal and

   b) $(\mathsf{F}\varphi[t])|_{1p} = \mathsf{F}(\varphi[t]|_p)$ for $p \in \textsc{Pos}(\varphi)$ if $\varphi|_p$ is a literal,

7. $(\varphi_1 \mathsf{U} \varphi_2)[\psi]|_{1p} = (\varphi_1 \mathsf{P} \varphi_2)|_{1p} = \varphi_1|_p$ for $p \in \textsc{Pos}(\varphi_1)$ and

8. $(\varphi_1 \mathsf{U} \varphi_2)[\psi]|_{2p} = (\varphi_1 \mathsf{P} \varphi_2)|_{2p} = \varphi_2|_p$ for $p \in \textsc{Pos}(\varphi_2)$.

Using replacement we define *reduction* formally as follows. Let a set $\{l_i \mapsto r_i \mid i = 1, \ldots, n\}$ of rules be given. Then

1. $\varphi \mapsto \psi$ if and only if there is $p \in \textsc{Pos}(\varphi)$, $i \in \{1, \ldots, n\}$ and a substitution σ such that $\sigma(l_i) = \varphi|_p$ and $\psi = \varphi[\sigma(r_i)]|_p$,

2. $\varphi \overset{n}{\mapsto} \psi$ for $n \in \{1, 2, \ldots\}$ if and only if there is a sequence $\varphi_0, \varphi_1, \ldots, \varphi_n$ such that $\varphi_0 = \varphi$, $\varphi_n = \psi$ and $\varphi_i \mapsto \varphi_{i+1}$ for $i \in \{0, 1, \ldots, n-1\}$ and

3. $\varphi \overset{*}{\mapsto} \psi$ if and only if $\varphi = \psi$ or there is an n such that $\varphi \overset{n}{\mapsto} \psi$.

A rewrite system given by a set of rules $\{l_i \mapsto r_i \mid i = 1, \ldots, n\}$ (for some finite n) is called *terminating* if the length of reductions is finite. Formally: there is no sequence $(\varphi_i)_{i \in \mathbb{N}}$ such that $\varphi_i \neq \varphi_{i+1}$ and $\varphi_i \mapsto \varphi_{i+1}$ for every $i \geq 0$. So in a terminating rewrite system every literal will be reduced to some literal which cannot be reduced any further. Similarly we will call literals φ_1 and φ_2 *joinable* if there is ψ such that $\varphi_1 \overset{*}{\mapsto} \psi$ and $\varphi_2 \overset{*}{\mapsto} \psi$. $\mapsto$ is called *confluent* (*locally confluent*) if for every φ such that $\varphi \overset{*}{\mapsto} \varphi_1$ ($\varphi \mapsto \varphi_1$) and $\varphi \overset{*}{\mapsto} \varphi_2$ ($\varphi \mapsto \varphi_2$) there is ψ such that $\varphi_1 \overset{*}{\mapsto} \psi$ and $\varphi_2 \overset{*}{\mapsto} \psi$. Termination and confluence are properties of a rewrite system which are essential if one wants to compute normal forms.

In order to analyze the confluence of a rewrite system it suffices to concentrate on a finite set of reductions, so called *critical pairs.* Assume that $r^{(1)} = l_{i_1} \mapsto r_{i_1}$ and $r^{(2)} = l_{i_2} \mapsto r_{i_2}$ are two rules. We will call $r^{(1)}$ and $r^{(2)}$ *overlapping (at position p)* if and only if there is a position $p \in \textsc{Pos}(l_{i_1})$ and a substitution σ such that $\sigma = \mathrm{mgu}(l_{i_1}|_p, l_{i_2})$.

Critical pairs are pairs of literals which can be derived from the application of a restricted kind of overlapping rules.

Definition 5.2.3 (Critical Pair, e.g. Baader and Nipkow [13])
Let $r^{(1)} = l_{i_1} \mapsto r_{i_1}$ and $r^{(2)} = l_{i_2} \mapsto r_{i_2}$ be overlapping at position $p \in \mathrm{Pos}(l_{i_1})$ such that $l_{i_1}|_p \notin \mathcal{X}$ and let $\sigma = \mathrm{mgu}(l_{i_1}|_p, l_{i_2})$. Then $\langle \sigma(r_{i_1}), \sigma(l_{i_1}[\sigma(r_{i_2})]|_p\rangle$ is called a *critical pair* (with respect to $\mapsto$).

The following classical result shows that confluence of a terminating rewrite system $\mapsto$ can be proved by checking if critical pairs are joinable.

Lemma 5.2.4 (Critical–Pair–Lemma, Knuth and Bendix [90])
A terminating rewrite system $\mapsto$ is confluent if and only if all critical pairs (with respect to $\mapsto$) are joinable.

From now on we will concentrate on the rewrite system described in Figure 5.1. This system will therefore be denoted as $\mapsto$. We will see that $\mapsto$ indeed has the desired properties. The first property is immediate.

Lemma 5.2.5
$\mapsto$ is terminating.

Proof. First observe that no application of a rewrite step yields a literal which is longer than the original literal. In particular there are several rules which shorten the literals. Since every literal consists of a finite number of symbols, these rules can only be applied a finite number of times. So now consider the *length–preserving* rules $\mathtt{Gnot}(\varphi) \mapsto \mathtt{not}(\mathsf{F}\varphi)$, $\mathtt{Fnot}(\varphi) \mapsto \mathtt{not}(\mathsf{G}\varphi)$ and $\mathtt{Xnot}(\varphi) \mapsto \mathtt{not}(\mathsf{X}\varphi)$. It is obvious that these rules push negations to the left. But this can also be done only a finite number of times (due to the finite length of literals), so the rules cannot be applied infinitely often. This proves the lemma. $\square$

In order to show that $\mapsto$ is also confluent we will prove that all critical pairs are joinable.

1. Let the rule $\mathsf{GG}\varphi \mapsto \mathsf{G}\varphi$ overlap with itself. Then there are two possible rewrite steps (the replaced subliteral is underlined):

$$\begin{aligned} \mathsf{G}\underline{\mathsf{GG}\varphi} &\mapsto \mathsf{GG}\varphi \text{ and}\\ \underline{\mathsf{GGG}\varphi} &\mapsto \mathsf{GG}\varphi. \end{aligned}$$

   Since the literals which emerge from applying these two steps are identical, this pair is clearly joinable.

2. Similarly it can be shown that an overlapping of $\mathsf{FF}\varphi \mapsto \mathsf{F}\varphi$ with itself is joinable using the following steps:

$$\begin{aligned} \mathsf{F}\underline{\mathsf{FF}\varphi} &\mapsto \mathsf{FF}\varphi \text{ and}\\ \underline{\mathsf{FFF}\varphi} &\mapsto \mathsf{FF}\varphi. \end{aligned}$$

3. If $\mathsf{G}\mathtt{not}(\varphi) \mapsto \mathtt{not}(\mathsf{F}\varphi)$ overlaps with $\mathtt{not}(\mathtt{not}(\varphi)) \mapsto \varphi$ we have

$$\begin{aligned} \mathsf{G}\underline{\mathtt{not}(\mathtt{not}(\varphi))} &\mapsto \mathsf{G}\varphi \text{ and}\\ \underline{\mathsf{G}\mathtt{not}(\mathtt{not}(\varphi))} &\mapsto \mathtt{not}(\underline{\mathsf{F}\mathtt{not}(\varphi)}) \mapsto \underline{\mathtt{not}(\mathtt{not}\mathsf{G}\varphi))} \mapsto \mathsf{G}\varphi. \end{aligned}$$

4. If $\mathsf{F}\mathtt{not}(\varphi) \mapsto \mathtt{not}(\mathsf{G}\varphi)$ overlaps with $\mathtt{not}(\mathtt{not}(\varphi)) \mapsto \varphi$ we have

$$\begin{aligned} \mathsf{F}\underline{\mathtt{not}(\mathtt{not}(\varphi))} &\mapsto \mathsf{F}\varphi \text{ and}\\ \underline{\mathsf{F}\mathtt{not}(\mathtt{not}(\varphi))} &\mapsto \mathtt{not}(\underline{\mathsf{G}\mathtt{not}(\varphi)}) \mapsto \underline{\mathtt{not}(\mathtt{not}(\mathsf{F}\varphi))} \mapsto \mathsf{F}\varphi. \end{aligned}$$

5. If $\mathtt{Xnot}(\varphi) \mapsto \mathtt{not}(\mathsf{X}\varphi)$ overlaps with $\mathtt{not}(\mathtt{not}(\varphi)) \mapsto \varphi$ we have

$$\begin{array}{rcl}\underline{\mathtt{X}\underline{\mathtt{not}(\mathtt{not}(\varphi))}} & \mapsto & \mathsf{X}\varphi \text{ and} \\ \underline{\mathtt{Xnot}(\mathtt{not}(\varphi))} & \mapsto & \mathtt{not}(\mathtt{X}\underline{\mathtt{not}(\varphi)}) \mapsto \underline{\mathtt{not}(\mathtt{not}(\mathsf{X}\varphi))} \mapsto \mathsf{X}\varphi.\end{array}$$

The other rules of the rewrite system can only overlap in a noncritical way, so each critical pair with respect to $\mapsto$ is joinable and the following lemma is proved.

Lemma 5.2.6
$\mapsto$ is confluent.

From now on let RED(φ) denote the uniquely determined reduced literal which emerges from the application of the above rewrite system. We have

$$\varphi \equiv \text{RED}(\varphi)$$

(due to the fact that $\mapsto$ is constructed from a set of semantical equivalences), and therefore the following theorem holds.

Theorem 5.2.2
For every FOLTL–literal φ there is a uniquely determined normal form RED(φ) with $\varphi \equiv \text{RED}(\varphi)$ which can be effectively computed.

In some of the following chapters we will restrict our analysis to reduced literals since these can be handled much easier than general ones.

5.2.3. Semantics for programs

We will now show how the semantics of a FOLTL–program P can be characterized in terms of the *stable model semantics* introduced by Gelfond and Lifschitz in [73]. The adaption of the Gelfond–Lifschitz constructions is necessary since rules in PROLOG(+T)–programs may contain negated literals in the tail (note that the problem of inducing stable models, i.e. models for normal (nontemporal) logic programs has been already addressed, i.e. in [128] and [125]). So the results from logic programming regarding definite logic programs such as the existence of monotonic and continuous operators which have fixpoints which can be seen as *least models* (in fact they are identical to the intersection of all Herbrand–models, see [164]) cannot be easily generalized to PROLOG(+T)–programs anymore. For nontemporal logic programming languages, Gelfond and Lifschitz have extended the classical *immediate consequence operator* T_P which characterizes the least Herbrand–Model to an operator which, given any subset M of B_P, computes a least Herbrand–model of a modified definite program P_M which (if M is chosen in the *right* way) has the property of being a *stable model* of P (see [73]).

We will now adapt the necessary concepts introduced in [73] in order to deal with PROLOG(+T)–programs.

So assume that a set P consisting of reduced ground rules is given and $M \subseteq B_P^{\text{FoLtl}}$ is any set of reduced ground atoms built from symbols occurring in P. The program P_M is then constructed as follows:

1. if there is a rule in P such that `not`(φ) for some $\varphi \in M$ occurs in the tail of this rule, then this rule is deleted and

2. negated atoms in the tails of the remaining clauses are also deleted.

Then P_M is clearly negation–free in the sense that no rule in P_M contains a negated atom in its tail. We will see that negation–free sets of PROLOG(+T)–ground–rules have models (in the case of satisfiability). Note that P is in general *not* satisfiable as shown in the example after Definition 5.2.2 on page 71.

For satisfiable negation–free programs we can indeed extend the fixpoint semantics of first order logic programming languages in a straightforward way. Recall that P has been required to consist only of ground rules, that is in general, P is not a finite set of rules anymore. We proceed as follows:

1. starting from the empty set $\emptyset$ we will construct an infinite tree consisting of nodes labeled with subsets of B_P^{FoLtl} and

2. show that the union of all labels of nodes on any maximal path in this tree is a set–based Herbrand–model of P.

We will use the following extension of the *immediate consequence operator* T_P (see [164] or [105]). This operator is used in the theory of first order Logic Programming to derive characterizations for the semantics of a definite program. The semantics of a program P is there given as the *smallest* set of ground atoms of the underlying first order language which P satisfies. Equivalently the semantics is given as the set of all implied ground atoms implied by the program. This set can be characterized as the least fixpoint of an operator (namely the operator T_P as introduced in [164]) and is easily seen to be uniquely determined (since Herbrand–models of definite logic programs can be intersected yielding Herbrand–models). For characterizing the semantics of PROLOG(+T)–programs we will change the original operator T_P in a way that allows the treatment of FoLtl–objects rather than only first order objects.

Definition 5.2.4 (Immediate Consequence Operator for FoLtl)
Let P be any negation–free set of ground rules. The mapping $T_P^{\text{FoLtl}} : 2^{B_P^{\text{FoLtl}}} \to 2^{B_P^{\text{FoLtl}}}$ is defined as follows:

$$T_P^{\text{FoLtl}}(I) = \left\{ \varphi \in B_P^{\text{FoLtl}} \mid \begin{array}{c} \text{there is a rule } \varphi :\!-\psi_1, \ldots, \psi_k. \text{ in } P \text{ such that} \\ I \models \psi_1, \ldots, \psi_k \end{array} \right\}.$$

This operator will be used together with the operator TEMPCLOSURE in alternating order.

Now assume that any negation–free PROLOG(+T)–program P which only contains ground rules is given. We construct an infinite labeled tree $T(P) = (V, E, l)$ with $l : V \to 2^{B_P^{\text{FoLtl}}}$ such that the following conditions are satisfied:

1. V contains a uniquely determined root node v_0,
2. $l(v_0) = \emptyset$ and
3. for each $v \in V$ it holds that
   a) if v is on an even level then there is $v' \in V$ such that $(v, v') \in E$ and $l(v') = T_P^{\text{FoLtl}}(l(v))$ and
   b) if v is on an odd level then assume that

   $$\text{TEMPCLOSURE}(l(v)) = \{\mathcal{T}_1, \ldots, \mathcal{T}_i, \ldots\}$$

   and there are nodes $v'_1, \ldots, v'_i, \cdots \in V$ such that $(v, v'_i) \in E$ and $l(v'_i) = \mathcal{T}_i$ for every $i \geq 1$.

Given $T(P)$ we define the set $p(T(P))$ to consist of all maximal paths in T, that is of all (in general infinite) maximal sequences of nodes $(v_i)_{i\in\mathbb{N}}$ such that $(v_i, v_{i+1}) \in E$. If $\pi \in p(T(P))$ is a maximal path then $V(\pi)$ denotes the set of nodes which are visited while traversing π.

Lemma 5.2.7

For every satisfiable set P of negation–free ground rules and for every $\pi \in p(T(P))$ it holds that

1. $\bigcup_{v\in V(\pi)} l(v) \models P$ and
2. $\bigcup_{v\in V(\pi)} l(v) \models P$ is temporally closed.

Proof. The first claim is clear by definition of T_P^{FoLtl}. For the proof of the second claim fix any $\pi \in p(T(P))$. If $\bigcup_{v\in V(\pi)} l(v)$ is not temporally closed, then an application of the operator TEMPCLOSURE would yield a temporally closed superset which shows that π is not maximal in this case. This contradicts the assumption on π. So $\bigcup_{v \in V(\pi)} l(v)$ is temporally closed. $\square$

So the models of P derived by the above construction can be collected in a set Θ_P as follows:

$$\Theta_P = \left\{ \bigcup_{v \in V(\pi)} l(v) \mid \pi \in p(T(P)) \right\}.$$

In order to deal with programs containing negation, we will now concentrate on programs P_M as described above.

Lemma 5.2.8

Let P be a set of ground PROLOG(+T)–rules and let $M \subseteq B_P^{\text{FoLtl}}$ be a set of atoms occurring in P. If P is satisfiable, then every $\mathcal{J} \in \Theta_{P_M}$ is a Herbrand–model of P.

Proof. Let M be as required. Then $\mathcal{J} \models P_M$ for every $\mathcal{J} \in \Theta_{P_M}$. Assume that $P_M = \{P_1, \ldots, P_n\}$, fix $\mathcal{J} \in \Theta_{P_M}$ and $i \in \{1, \ldots, n\}$. Since $\mathcal{J} \models P_M$ we have $\mathcal{J} \models P_i$. Consider the following cases.

Case 1 $P_i \in P$. Then the claim is immediate.

Case 2 $P_i \notin P$. Then P_i emerges from a rule from P by deleting negated atoms in the tail of P_j. That is for $P_i = \varphi :\!- \psi_1, \ldots, \psi_k$. there is $P_j = \varphi :\!- \psi_1, \ldots, \psi_k, \bar{\psi}_1, \ldots, \bar{\psi}_l$. in P for $\bar{\psi}_1, \ldots, \bar{\psi}_l \in B_P^{\text{FoLtl}}$. Since $\mathcal{J} \models P_i$ we have $\mathcal{J} \models \varphi \vee \neg\psi_1, \ldots, \neg\psi_k$ which implies $\mathcal{J} \models \varphi \vee \neg\psi_1 \vee \cdots \vee \neg\psi_k \vee \bar{\psi}_1 \vee \cdots \vee \bar{\psi}_l$ and therefore $\mathcal{J} \models P_j$.

It remains to prove that also the rules C from P which have been deleted during the construction of P_M are satisfied by $\mathcal{J}$. If C is such a rule, then $C = \varphi :\!- \psi_1, \ldots, \psi_k, \texttt{not}(\psi)$. for some $\psi \in M$. Since $\mathcal{J} \models M$ the claim is immediate. So $\mathcal{J}$ is a model of P and the lemma is proved. $\square$

5.3. A Saturation–based temporal Proof Procedure

As we have already mentioned in chapter 5.2 we will now study the *operational semantics* of PROLOG(+T). This semantics will be studied by presenting a calculus which is both sound and refutation complete. Basically this calculus will be an extension of the well–known *tableaux* principle which is a quite popular principle in many areas of automated reasoning. In contrast to PROLOG where an eventually modified (and speed up) version of the *SLD–resolution* principle is chosen for proving logical consequence of goals (and therefore for computing answers to programs) resolution–based approaches are not the best choice for our problem. Consider the following problem: given a program $P = \{P_1, \ldots, P_n\}$ and a goal $G =:- \varphi_1, \ldots, \varphi_m$. Assume that i is such that $\varphi_i = \mathtt{Gp(X)}$. Then $\varphi_i \equiv \mathtt{p(X)} \wedge \mathtt{XGp(X)}$. If there is for example a rule $P_j = \mathtt{XXXp(X)}\ :- \mathtt{q(Y,Y)}.$ in P the *normal* SLD–principle will not be able to compute a resolvent. But it is obvious that $G' = \varphi_1, \ldots, \varphi_{i-1}, \mathtt{p(X)}, \mathtt{Xp(X)}, \mathtt{XXp(X)}, \mathtt{q(Y,Y)}, \mathtt{XXXXGp(X)}, \varphi_{i+1}, \ldots, \varphi_m$ should be a resolvent of G and P_j. Therefore we will adapt the tableaux calculus in order to deal with temporal constructs. We will see that the resulting proof procedure is indeed sound and (in some sense) refutation complete.

The study of sound and complete proof–procedures for FOLTL and its fragments dates as far back as to the first contributions of Abadi and Manna (see [3]). It has been pointed out quite early that the whole logic FOLTL is not only undecidable (which is clear since it contains the whole first order logic) but also not recursively enumerable (see e.g. [82] or [30]). Consequently no complete proof–procedure for FOLTL can exist. But restrictions of FOLTL yield recursively enumerable fragments. For example in [82] a restricted usage of temporal operators yields the *monodic fragment* which can be embedded into the monadic second order theory[5]. Omitting the use of U and P yields a logic which can be recursively enumerated. Enumeration can be carried out by a sequent–style calculus (see

[5]A formula φ is called *monodic* if every subformula ψ of φ which has the form $\psi = \psi_1 \oplus \psi_2$ for $\oplus \in \{\mathsf{U}, \mathsf{P}\}$ has at most one free variable. Consequently the *monodic fragment* of first order tempporal logic consists of all monodic formulas.

[14]). Other types of proof–procedures for FoLtl–fragments are Hilbert–style calculi (see e.g. [53]), Resolution–style calculi (see [91], [41], [43], [44], [91], [41] and [42]) and ASM–based procedures (see [174]). A good survey of first order temporal logic proof–procedures has been presented recently by C. Castellini in his Ph.D. Thesis (see [27]).

Our proof–procedure which we will present on this section is a tableaux–style procedure. Several modifications of the tableaux–construction for first order logic have been presented in order to derive proof–procedures for first order temporal logic languages (see [28] and [29] for FoLtl and [114] for the first order version of Ctl). Nevertheless our method has its justification since it is defined especially for Prolog(+T)–formulas and since it integrates the reduction–concept for temporal literals introduced above.

5.3.1. Tableaux Rules

Since Prolog(+T) is first order logic based we will have to define a proof procedure which enables us to handle first order logic constructs. The tableaux method (see [66]) is one such method. Basically it constructs a tableaux (which is represented as a directed graph) consisting of nodes which are labeled with sets of formulas. The key property for the soundness and completeness is the following: if n_1 and n_2 are such nodes and F_1 and F_2 are the sets of formulas labeling these two nodes and n_2 is a successor of n_1 then unsatisfiability of F_2 implies unsatisfiability of F_1.

We will distinguish two principle kinds of tableaux rules here: *expansion rules* and *saturation rules*. Expansion rules are rules which consider the *first order part* of a formula. An example might be the following: if $\varphi_1 \wedge \varphi_2$ is to be satisfied, then *both* φ_1 and φ_2 have to be satisfied at the same time. On the other hand, saturation rules consider the *temporal part* of formulas. Here we can argue with the example presented above: if $\mathsf{G}\varphi$ is to be satisfied, then φ and $\mathsf{XG}\varphi$ have to be satisfied (in other words $\mathsf{X}^i\varphi$ has to be satisfied for *every* $i \geq 0$).

The discussion of the proof procedure will now proceed as follows: first we will have to define certain concepts which are needed for the presentation of the proof procedure. This will include a node format for the nodes of the tableaux. After having introduced the basics we will consider the expansion rules and the saturation rules separately. We will see that the expansion rules are the rules from the *classical* tableaux calculus for first order predicate logic (moreover, the proof procedure to be introduced is capable of handling formulas which are not restricted to the formulas occurring in PROLOG(+T)–programs). In order to be applicable we will then present two kinds of termination criteria: criteria for termination in the case of success and in the case of failure. We will see that termination by failure is a much more difficult task than successful termination. The chapter will then be continued by proving the soundness and refutation completeness of the procedure.

Basic Formalisms

We will now describe the format of the nodes from which a tableaux is built up.

Definition 5.3.1 (Tableaux node)
A *tableaux node* is a tuple $n = (\Phi, F, S)$ where Φ is a nonempty set of FOLTL–formulas and F and S are (possibly empty) sets of tableaux nodes. The sets F and S will be referred to as the *father nodes* and the *son nodes* of n.

For the proofs in later sections we will introduce the following abbreviation. For a node n we will denote the set of formulas *labeling* this node by $\mathcal{F}(n)$. In other words if $n = (\Phi, F, S)$, then $\mathcal{F}(n) = \Phi$.

Definition 5.3.2 (Path)
A *path* is a sequence $\pi = (n_0, n_1, \ldots, n_k, \ldots)$ of tableaux nodes such that $n_i = (\Phi_i, F_i, S_i)$ for $i = 0, 1, \ldots,$ $F_0 = \emptyset$ and for each $i \in \{0, 1, \ldots\}$ it holds that

$n_{i+1} \in S_i$. A node n is said to *occur* on π if there is an index i such that $n_i = n$. If $\pi = (n_0, n_1, \ldots, n_{k-1})$, the integer k is called the *length* of π and π is called *finite* (or *of finite length*). Otherwise (i.e. if no such k exists), π is called *infinite* (or *of infinite length*).

Note that in general paths in a tableaux will be of infinite length. This will be of interest in the proof of refutation completeness which we will present later on.

We will now describe the construction of the *initial tableaux node* for the tableaux to be constructed from a formula φ. This node is given as follows:

$$n_0 = (\{\varphi\}, \emptyset, \emptyset)\,.$$

Whenever we will refer to the *initial node* of a tableaux from now on we will mean a node n_0 constructed in this way.

Expansion Rules

We will now describe how to construct new nodes from nodes which have already been constructed. The rules to be described in this section will later on be referred to as *expansion rules*. We assume that $n = (\Phi, F, S)$ is a node which has been chosen from a set N of nodes. A fixed selection rule has to be chosen in order to select formulas from $\mathcal{F}(n)$ to which the rules can be applied. The only requirement which we will have to put on the selection rule is the requirement of *fairness*. Informally *fairness* means that no application of a rule to a formula from $\mathcal{F}(n)$ is retarded for an infinite time. Equivalently we can say that every possible inference step is eventually carried out (as long as the procedure does not terminate).

Now let φ be the selected formula. We distinguish the following cases:

1. $\varphi = \varphi_1 \wedge \varphi_2$. In this case we create a node $n' = (\mathcal{F}(n'), F', S')$ as follows:

   $$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\varphi_1 \wedge \varphi_2\}) \cup \{\varphi_1, \varphi_2\},$$

   $F' = \{n\}$, $S' = \emptyset$. Furthermore we update the set S of sons of n to $S = S \cup \{n\}$. This rule will from now on be referred to as ($\wedge$**1**).

2. $\varphi = \neg(\varphi_1 \vee \varphi_2)$. In this case we create a node $n' = (\mathcal{F}(n'), F', S')$ as follows:

   $$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\neg(\varphi_1 \wedge \varphi_2)\}) \cup \{\neg\varphi_1, \neg\varphi_2\},$$

   $F' = \{n\}$, $S' = \emptyset$. Furthermore we update the set S of sons of n to $S = S \cup \{n\}$. This rule will from now on be referred to as ($\wedge$**2**).

3. $\varphi = \varphi_1 \vee \varphi_2$. In this case we create two nodes $n' = (\mathcal{F}(n'), F', S')$ and $n'' = (\mathcal{F}(n''), F'', S'')$ as follows:

   $$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\varphi_1 \vee \varphi_2\}) \cup \{\varphi_1\},$$

   $F' = \{n\}$, $S' = \emptyset$ and

   $$\mathcal{F}(n'') = (\mathcal{F}(n) \setminus \{\varphi_1 \vee \varphi_2\}) \cup \{\varphi_2\},$$

   $F''\{n\} =$, $S' = \emptyset$. The set S of sons of n will be updated to $S = S \cup \{n', n''\}$. This rule will be referred to as ($\vee$**1**).

4. $\varphi = \neg(\varphi_1 \wedge \varphi_2)$. In this case we create two nodes $n' = (\mathcal{F}(n'), F', S')$ and $n'' = (\mathcal{F}(n''), F'', S'')$ as follows:

   $$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\neg(\varphi_1 \wedge \varphi_2)\}) \cup \{\neg\varphi_1\},$$

$F' = \{n\}$, $S' = \emptyset$ and

$$\mathcal{F}(n'') = (\mathcal{F}(n) \setminus \{\neg(\varphi_1 \wedge \varphi_2)\}) \cup \{\neg\varphi_2\},$$

$F''\{n\} =$, $S' = \emptyset$. The set S of sons of n will be updated to $S = S \cup \{n', n''\}$. This rule will be referred to as ($\vee$**2**).

5. $\varphi = \varphi_1 \rightarrow \varphi_2$. In this case two nodes $n' = (\mathcal{F}(n'), F', S')$ and $n'' = (\mathcal{F}(n''), F'', S'')$ will be constructed with

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\varphi_1 \rightarrow \varphi_2\}) \cup \{\neg\varphi_1\},$$

$F' = \{n\}$, $S' = \emptyset$ and

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\varphi_1 \rightarrow \varphi_2\}) \cup \{\varphi_2\},$$

$F' = \{n\}$, $S' = \emptyset$. The set S will be updated to $S = S \cup \{n', n''\}$. This rule will be referred to as ($\rightarrow$ **1**).

6. $\varphi = \neg(\varphi_1 \rightarrow \varphi_2)$. Here a node $n' = (\mathcal{F}(n'), F', S')$ will be constructed with

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\neg(\varphi_1 \rightarrow \varphi_2)\}) \cup \{\varphi_1, \neg\varphi_2\},$$

$F' = \{n\}$ and $S' = \emptyset$. S will be updated to $S = S \cup \{n'\}$. The rule will be referred to as ($\rightarrow$ **2**).

7. $\varphi = \varphi_1 \leftrightarrow \varphi_2$. Here we create a node $n' = (\mathcal{F}(n'), F', S')$ with

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\varphi_1 \leftrightarrow \varphi_2\}) \cup \{\varphi_1 \rightarrow \varphi_2, \varphi_2 \rightarrow \varphi_1\},$$

$F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n'\}$. The rule will be referred to as ($\leftrightarrow$ **1**).

8. $\varphi = \neg(\varphi_1 \leftrightarrow \varphi_2)$. Here we create two nodes $n' = (\mathcal{F}(n'), F', S')$ and $n'' = (\mathcal{F}(n''), F'', S'')$ with

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\neg(\varphi_1 \leftrightarrow \varphi_2)\}) \cup \{\neg(\varphi_1 \rightarrow \varphi_2)\},$$

   $F' = \{n\}$, $S' = \emptyset$ and

$$\mathcal{F}(n'') = (\mathcal{F}(n) \setminus \{\neg(\varphi_1 \leftrightarrow \varphi_2)\}) \cup \{\neg(\varphi_2 \rightarrow \varphi_1)\},$$

   $F'' = \{n\}$, $S'' = \emptyset$. Again S is updated to $S = S \cup \{n', n''\}$. The rule will be referred to as ($\leftrightarrow$ **2**).

9. $\varphi = \neg\neg\psi$. Here we will create a node $n' = (\mathcal{F}(n'), F', S')$ with

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\neg\neg\psi\}) \cup \{\psi\},$$

   $F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n'\}$ and the rule is referred to as ($\neg\neg$**E**).

10. $\varphi = \neg\forall \mathtt{X}\psi(\mathtt{X})$. Here we will create a node $n' = (\mathcal{F}(n'), F', S')$ with

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\neg\forall \mathtt{X}\psi(\mathtt{X})\}) \cup \{\exists \mathtt{X}\neg\psi(\mathtt{X})\},$$

   $F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n'\}$ and the rule is referred to as (**R**$\forall$).

11. $\varphi = \neg\exists \mathtt{X}\psi(\mathtt{X})$. Here we will create a node $n' = (\mathcal{F}(n'), F', S')$ with

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\neg\exists \mathtt{X}\psi(\mathtt{X})\}) \cup \{\forall \mathtt{X}\neg\psi(\mathtt{X})\},$$

$F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n'\}$ and the rule is referred to as $(\mathbf{R}\exists)$.

12. $\varphi = \forall \mathtt{X}\psi(\mathtt{X})$. Here we create a new node $n' = (\mathcal{F}(n'), F', S')$ with

$$\mathcal{F}(n') = \mathcal{F}(n) \cup \{\psi(t)\}$$

for *some* ground term t which can be built up from symbols occurring in $\mathcal{F}(n)$, $F' = \{n\}$ and $S' = \emptyset$. We update S to $S = S \cup \{n'\}$ and refer to this rule as $(\sigma\text{–}\mathbf{I_1})$.

13. $\varphi = \exists \mathtt{X}\psi(\mathtt{X})$. Here we create a new node $n' = (\mathcal{F}(n'), F', S')$ with

$$\mathcal{F}(n') = \mathcal{F}(n) \cup \{\psi(\mathtt{X}_{\text{new}})\}$$

for *some* variable symbol $\mathtt{X}_{\text{new}} \notin \textsc{Var}(\mathcal{F}(n))$, $F' = \{n\}$ and $S' = \emptyset$. We update S to $S = S \cup \{n'\}$ and refer to this rule as $(\sigma\text{–}\mathbf{I_2})$.

14. $\varphi = \forall \mathtt{X}\psi$ and $\mathtt{X} \notin \textsc{Var}(\psi)$. Here we create a node $n' = (\mathcal{F}(n'), F', S')$ with

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\forall \mathtt{X}\psi\}) \cup \{\psi\},$$

$F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n'\}$ and the rule is referred to as $(\forall\text{–}\mathbf{E})$.

15. $\varphi = \exists \mathtt{X}\psi$ and $\mathtt{X} \notin \textsc{Var}(\psi)$. Here we create a node $n' = (\mathcal{F}(n'), F', S')$ with

$$\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\exists \mathtt{X}\psi\}) \cup \{\psi\},$$

$F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n'\}$ and the rule is referred to as $(\exists\text{–}\mathbf{E})$.

We can group the above rules as follows: $(\wedge\mathbf{1})$ and $(\wedge\mathbf{2})$ are called $\wedge$*–type rules*, $(\vee\mathbf{1})$

and ($\vee$**2**) are called $\vee$*–type rules*, ($\rightarrow$ **1**) and ($\rightarrow$ **2**) are called $\rightarrow$*–type rules* and ($\leftrightarrow$ **1**) and ($\leftrightarrow$ **2**) are called $\leftrightarrow$*–type rules*. Furthermore ($\neg$–**E**) is called the $\neg$*–elimination rule*, (**R**$\forall$) and (**R**$\exists$) are called *rewrite rules*, (σ–$\mathbf{I_1}$) and (σ–$\mathbf{I_2}$) are called *substitution rules* and ($\forall$–**E**) and ($\exists$–**E**) are called *quantifier–elimination rules*. The rules are summarized in Figure 5.2. Here the formula above the fraction stroke denotes the selected formula from $\mathcal{F}(n)$ and the formulas below the fraction strokes denote the formulas created from this formula. The number of columns below the fraction stroke denotes the number of new nodes to be created. For example in

$$\frac{\varphi_1 \wedge \varphi_2}{\begin{matrix}\varphi_1 \\ \varphi_2\end{matrix}}$$

one new node containing two new formulas has to be created.

Saturation Rules

In contrast to the expansion rules described above, the saturation rules which will be described deal with the temporal part of a formula rather than with the first order part. We will present two rules for each of the operators G, F, U and P. For the operator X no such rule will be given.

As for the expansion rules we will describe the rules separately by distinguishing the different possibilities of how a selected formula might look like. For the rest of this section assume that i is any *fixed* integer and $n = (\mathcal{F}(n), F, S)$ is the node from which a formula φ is chosen. The saturation rules are given as follows:

1. If $\varphi = \mathsf{X}^i\mathsf{G}\psi$, then we create a new node $n' = (\mathcal{F}(n'), F', S')$ with $\mathcal{F}(n') = \left(\mathcal{F}(n) \setminus \{\mathsf{X}^i\mathsf{G}\psi\}\right) \cup \{\mathsf{X}^i\psi, \mathsf{X}^{i+1}\mathsf{G}\psi\}$, $F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n\}$. The rule is referred to as ($\mathsf{G1}$).

$$(\wedge\mathbf{1})\quad \frac{\varphi_1 \wedge \varphi_2}{\begin{array}{c}\varphi_1\\ \varphi_2\end{array}} \qquad (\wedge\mathbf{2})\quad \frac{\neg(\varphi_1 \vee \varphi_2)}{\begin{array}{c}\neg\varphi_1\\ \neg\varphi_2\end{array}}$$

$$(\vee\mathbf{1})\quad \frac{\varphi_1 \vee \varphi_2}{\varphi_1 \quad \varphi_2} \qquad (\vee\mathbf{2})\quad \frac{\neg(\varphi_1 \wedge \varphi_2)}{\neg\varphi_1 \quad \neg\varphi_2}$$

$$(\rightarrow\mathbf{1})\quad \frac{\varphi_1 \rightarrow \varphi_2}{\neg\varphi_1 \quad \varphi_2} \qquad (\rightarrow\mathbf{2})\quad \frac{\neg(\varphi_1 \rightarrow \varphi_2)}{\begin{array}{c}\varphi_1\\ \neg\varphi_2\end{array}}$$

$$(\leftrightarrow\mathbf{1})\quad \frac{\varphi_1 \leftrightarrow \varphi_2}{\begin{array}{c}\varphi_1 \rightarrow \varphi_2\\ \varphi_2 \rightarrow \varphi_1\end{array}} \qquad (\leftrightarrow\mathbf{2})\quad \frac{\neg(\varphi_1 \leftrightarrow \varphi_2)}{\neg(\varphi_1 \rightarrow \varphi_2) \quad \neg(\varphi_2 \rightarrow \varphi_1)}$$

$$(\neg\text{–}\mathbf{E})\quad \frac{\neg\neg\varphi}{\varphi}$$

$$(\forall\text{–}\mathbf{E})\quad \frac{\forall\mathtt{X}\varphi}{\varphi}\quad \mathtt{X} \notin \textsc{Var}(\varphi) \qquad (\exists\text{–}\mathbf{E})\quad \frac{\exists\mathtt{X}\varphi}{\varphi}\quad \mathtt{X} \notin \textsc{Var}(\varphi)$$

$$(\mathbf{R}\forall)\quad \frac{\neg\forall\mathtt{X}\varphi(\mathtt{X})}{\exists\mathtt{X}\neg\varphi(\mathtt{X})} \qquad (\mathbf{R}\exists)\quad \frac{\neg\exists\mathtt{X}\varphi(\mathtt{X})}{\forall\mathtt{X}\neg\varphi(\mathtt{X})}$$

$$(\sigma\text{–}\mathbf{I_1})\quad \frac{\forall\mathtt{X}\varphi(\mathtt{X})}{\varphi(t)}\quad \text{for a ground term } t \text{ which can be built up from symbols occurring in } \mathcal{F}(n)$$

$$(\sigma\text{–}\mathbf{I_2})\quad \frac{\exists\mathtt{X}\varphi(\mathtt{X})}{\varphi(\mathtt{X}_{\text{new}})}\quad \text{for any } \textit{new} \text{ variable symbol } \mathtt{X}_{\text{new}}$$

Figure 5.2.: Expansion Rules

2. If $\varphi = \neg\mathsf{X}^i\mathsf{G}\psi$, then we create two new nodes $n' = (\mathcal{F}(n'), F', S')$ and $n'' = (\mathcal{F}(n''), F'', S'')$ with $\mathcal{F}(n') = \big(\mathcal{F}(n) \setminus \{\neg\mathsf{X}^i\mathsf{G}\psi\}\big) \cup \{\neg\mathsf{X}^i\psi\}$, $F' = \{n\}$ and $S' = \emptyset$ and $\mathcal{F}(n'') = \big(\mathcal{F}(n) \setminus \{\neg\mathsf{X}^i\mathsf{G}\psi\}\big)\cup\{\neg\mathsf{X}^{i+1}\mathsf{G}\psi\}$, $F'' = \{n\}$ and $S'' = \emptyset$. Additionally S is updated to $S = S \cup \{n', n''\}$. The rule will be called ($\mathsf{G}\mathbf{2}$).

3. If $\varphi = \mathsf{X}^i\mathsf{F}\psi$, then we create two new nodes $n' = (\mathcal{F}(n'), F', S')$ and $n'' = (\mathcal{F}(n''), F'', S'')$ with $\mathcal{F}(n') = \big(\mathcal{F}(n) \setminus \{\mathsf{X}^i\mathsf{F}\psi\}\big) \cup \{\mathsf{X}^i\psi\}$, $F' = \{n\}$ and $S' = \emptyset$ and $\mathcal{F}(n'') = \big(\mathcal{F}(n) \setminus \{\mathsf{X}^i\mathsf{F}\psi\}\big) \cup \{\mathsf{X}^{i+1}\mathsf{F}\psi\}$, $F'' = \{n\}$ and $S'' = \emptyset$. Additionally S is updated to $S = S \cup \{n', n''\}$. The rule will be called ($\mathsf{F}\mathbf{1}$).

4. If $\varphi = \neg\mathsf{X}^i\mathsf{F}\psi$, then we create a new node $n' = (\mathcal{F}(n'), F', S')$ with $\mathcal{F}(n') = \big(\mathcal{F}(n) \setminus \{\neg\mathsf{X}^i\mathsf{F}\psi\}\big) \cup \{\neg\mathsf{X}^i\psi, \neg\mathsf{X}^{i+1}\mathsf{F}\psi\}$, $F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n\}$. The rule is referred to as ($\mathsf{F}\mathbf{2}$).

5. If $\varphi = \mathsf{X}^i\psi_1\mathsf{U}\psi_2$, then we create two new nodes $n' = (\mathcal{F}(n'), F', S')$ and $n'' = (\mathcal{F}(n''), F'', S'')$ with $\mathcal{F}(n') = \big(\mathcal{F}(n) \setminus \{\mathsf{X}^i\psi_1\mathsf{U}\psi_2\}\big) \cup \{\mathsf{X}^i\psi_2\}$, $F' = \{n\}$ and $S' = \emptyset$ and $\mathcal{F}(n'') = \big(\mathcal{F}(n) \setminus \{\mathsf{X}^i\psi_1\mathsf{U}\psi_2\}\big) \cup \{\mathsf{X}^i\psi_1 \wedge \mathsf{X}^{i+1}\psi_1\mathsf{U}\psi_2\}$, $F'' = \{n\}$ and $S'' = \emptyset$. Additionally S is updated to $S = S \cup \{n', n''\}$. The rule will be called ($\mathsf{U}\mathbf{1}$).

6. If $\varphi = \neg\mathsf{X}^i\psi_1\mathsf{U}\psi_2$, then we create a new node $n' = (\mathcal{F}(n'), F', S')$ with $\mathcal{F}(n') = \big(\mathcal{F}(n) \setminus \{\neg\mathsf{X}^i\psi_1\mathsf{U}\psi_2\}\big) \cup \{\neg\mathsf{X}^i\psi_2, \neg\mathsf{X}^i\psi_1 \vee \neg\mathsf{X}^{i+1}\psi_1\mathsf{U}\psi_2\}$, $F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n\}$. The rule is referred to as ($\mathsf{U}\mathbf{2}$).

7. If $\varphi = \mathsf{X}^i\psi_1\mathsf{P}\psi_2$, then we create a new node $n' = (\mathcal{F}(n'), F', S')$ with $\mathcal{F}(n') = \big(\mathcal{F}(n) \setminus \{\mathsf{X}^i\psi_1\mathsf{P}\psi_2\}\big) \cup \{\neg\mathsf{X}^i\psi_2, \mathsf{X}^i\psi_1 \vee \mathsf{X}^{i+1}\psi_1\mathsf{P}\psi_2\}$, $F' = \{n\}$ and $S' = \emptyset$. S is updated to $S = S \cup \{n\}$. The rule is referred to as ($\mathsf{P}\mathbf{1}$).

8. If $\varphi = \neg\mathsf{X}^i\psi_1\mathsf{P}\psi_2$, then we create two new nodes $n' = (\mathcal{F}(n'), F', S')$ and $n'' = (\mathcal{F}(n''), F'', S'')$ with $\mathcal{F}(n') = \big(\mathcal{F}(n) \setminus \{\neg\mathsf{X}^i\psi_1\mathsf{P}\psi_2\}\big) \cup \{\mathsf{X}^i\psi_2\}$, $F' = \{n\}$ and $S' = \emptyset$ and $\mathcal{F}(n'') = \big(\mathcal{F}(n) \setminus \{\neg\mathsf{X}^i\psi_1\mathsf{P}\psi_2\}\big) \cup \{\neg\mathsf{X}^i\psi_1 \wedge \neg\mathsf{X}^{i+1}\psi_1\mathsf{P}\psi_2\}$, $F'' = \{n\}$ and

G–type

$$(\mathsf{G1})\ \frac{\mathsf{X}^i\mathsf{G}\varphi}{\begin{array}{c}\mathsf{X}^i\varphi\\ \mathsf{X}^{i+1}\mathsf{G}\varphi\end{array}} \qquad (\mathsf{G2})\ \frac{\neg\mathsf{X}^i\mathsf{G}\varphi}{\neg\mathsf{X}^i\varphi \quad \neg\mathsf{X}^{i+1}\mathsf{G}\varphi}$$

F–type

$$(\mathsf{F1})\ \frac{\mathsf{X}^i\mathsf{F}\varphi}{\mathsf{X}^i\varphi \quad \mathsf{F}^{i+1}\mathsf{F}\varphi} \qquad (\mathsf{F2})\ \frac{\neg\mathsf{X}^i\mathsf{F}\varphi}{\begin{array}{c}\neg\mathsf{X}^i\varphi\\ \neg\mathsf{X}^{i+1}\mathsf{F}\varphi\end{array}}$$

U–type

$$(\mathsf{U1})\ \frac{\mathsf{X}^i\varphi_1\mathsf{U}\varphi_2}{\mathsf{X}^i\varphi_2 \quad \mathsf{X}^i\varphi_1 \wedge \mathsf{X}^{i+1}\varphi_1\mathsf{U}\varphi_2} \qquad (\mathsf{U2})\ \frac{\neg\mathsf{X}^i\varphi_1\mathsf{U}\varphi_2}{\begin{array}{c}\neg\mathsf{X}^i\varphi_2\\ \neg\mathsf{X}^i\varphi_1 \vee \neg\mathsf{X}^{i+1}\varphi_1\mathsf{U}\varphi_2\end{array}}$$

P–type

$$(\mathsf{P1})\ \frac{\mathsf{X}^i\varphi_1\mathsf{P}\varphi_2}{\begin{array}{c}\neg\mathsf{X}^i\varphi_2\\ \mathsf{X}^i\varphi_1 \vee \mathsf{X}^{i+1}\varphi_1\mathsf{P}\varphi_2\end{array}} \qquad (\mathsf{P2})\ \frac{\neg\mathsf{X}^i\varphi_1\mathsf{P}\varphi_2}{\mathsf{X}^i\varphi_2 \quad \neg\mathsf{X}^i\varphi_1 \wedge \neg\mathsf{X}^{i+1}\varphi_1\mathsf{P}\varphi_2}$$

Reduction Rule

$$(\mathbf{Red})\ \frac{\varphi}{\textsc{Red}(\varphi)} \quad \text{if } \textsc{Red}(\varphi) \neq \varphi$$

Figure 5.3.: Saturation Rules

$S'' = \emptyset$. Additionally S is updated to $S = S \cup \{n', n''\}$. The rule will be called (P**2**).

Additionally we will have to use the following rule (called the *reduction rule*): If φ is a literal and $\textsc{Red}(\varphi) \neq \varphi$, then we create a new node $n' = (\mathcal{F}(n'), F', S')$ with $\mathcal{F}(n') = (\mathcal{F}(n) \setminus \{\varphi\}) \cup \{\textsc{Red}(\varphi)\}$, set $F' = \{n\}$ and $S' = \emptyset$ and update S to $S = S \cup \{n'\}$. As for the expansion rules the saturation rules are summarized in Figure 5.3.

Termination Criteria

In order to be useful the tableaux procedure needs some criteria to indicate when the expansion of a node can be aborted. This is a nontrivial task since in general (and in contrast to first order logic) there is *nearly always* a rule which may be applied to some formula. To be more precise if n is a tableaux node and if there is at least one formula $\varphi \in \mathcal{F}(n)$ such that φ contains one of the operators G, F, U and P, there is an infinite sequence $(n_i)_{i=j_0}^{\infty}$ of tableaux nodes such that $n_{j_0} = n$ and each $n_{j_{i+1}}$ emerges from n_{j_i} by application of a saturation rule. Consider for example a node n such that $\mathsf{G}\varphi \in \mathcal{F}(n)$ for some formula φ. Then a node n' can be constructed such that $\{\varphi, \mathsf{XG}\varphi\} \subseteq \mathcal{F}(n')$ (by application of the rule ($\mathtt{G1}$)). This rule can then be applied to n' yielding n'' such that $\{\mathsf{X}\varphi, \mathsf{XXG}\varphi\} \subseteq \mathcal{F}(n'')$ and so forth. This example might illustrate the need of some more sophisticated criteria of when to abort the expansion of nodes.

We will adapt the concept of *closedness* of a node which is known from first order logic to include the temporal operators. In first order logic a node is called *closed* if there are literals $\varphi_1, \varphi_2 \in \mathcal{F}(n)$ such that $\varphi_2 = \neg\psi$ and φ_1 and φ_2 are unifiable. For first order logical literals this is adequate but for FoLtl–literals we need a more complicated criterion since the syntactical form of a literal is in general not unique (consider e.g. $\varphi_1 = \mathtt{not}(\mathsf{X}p(\mathtt{X}))$ and $\varphi_2 = \mathsf{X}\mathtt{not}(\mathtt{p}(\mathtt{X}))$ where $\varphi_1 \neq \varphi_2$ but $\varphi_1 \equiv \varphi_2$).

Definition 5.3.3

Let n be a tableaux node. Then n is called *closed* if there are formulas φ_1 and φ_2 from $\mathcal{F}(n)$ such that

1. $\textsc{Red}(\varphi_1)$ and $\textsc{Red}(\neg\varphi_2)$ are unifiable or
2. there is an $i \geq 0$ such that $\textsc{Red}(\varphi_1) = \mathsf{X}^i\psi_1^{(1)}\mathsf{P}\psi_2^{(1)}$ and $\textsc{Red}(\varphi_2) = \mathsf{X}^i\psi^{(2)}$ and $\textsc{Red}(\psi_2^{(1)})$ and $\textsc{Red}(\psi^{(2)})$ are unifiable or
3. there is an $i \geq 0$ such that $\textsc{Red}(\varphi_1) = \neg\mathsf{X}^i\psi_1^{(1)}\mathsf{U}\psi_2^{(1)}$ and $\textsc{Red}(\varphi_2) = \mathsf{X}^i\psi^{(2)}$ and $\textsc{Red}(\psi_2^{(1)})$ and $\textsc{Red}(\psi^{(2)})$ are unifiable.

A path π is called *closed* if it contains a closed node. Otherwise π is called *open*.

The definition of closedness presented here is more complicated then necessary. It would suffice to define closedness solely by the first point of the definition. However, the two other points allow detection of closedness at an earlier point of time which might speed up the proof procedure.

Given the definition of closedness we are able to detect several nontrivial cases of closed nodes.

1. Assume that $\{\neg\mathsf{X}\mathsf{p}(\mathtt{X}), \neg\mathsf{X}\neg\mathsf{p}(\mathtt{X})\} \subseteq \mathcal{F}(n)$. Then $\textsc{Red}(\neg\mathsf{X}\mathsf{p}(\mathtt{X})) = \neg\mathsf{X}\mathsf{p}(\mathtt{X})$ and $\textsc{Red}(\neg\neg\mathsf{X}\neg\mathsf{p}(\mathtt{X})) = \neg\mathsf{X}\mathsf{p}(\mathtt{X})$ and an mgu is given by ε. So n is closed.

2. Assume that $\{\mathsf{GXF}\neg\mathsf{Gp}(\mathtt{X}, \mathsf{a}), \mathsf{FFXGGGp}(\mathsf{b}, \mathtt{Y}\} \subseteq \mathcal{F}(n)$. Then

$$\textsc{Red}(\mathsf{GXF}\neg\mathsf{Gp}(\mathtt{X}, \mathsf{a})) = \neg\mathsf{FXGp}(\mathtt{X}, \mathsf{a})$$

   and

$$\textsc{Red}(\neg\mathsf{FFXGGGp}(\mathsf{b}, \mathtt{Y})) = \neg\mathsf{FXGp}(\mathsf{b}, \mathtt{Y}).$$

   An mgu is given by $\{\frac{\mathtt{X}}{\mathsf{b}}, \frac{\mathtt{Y}}{\mathsf{a}}\}$. So n is closed.

3. Finally assume that $\{\mathsf{XXFp}(\mathsf{a})\mathsf{Pq}(\mathsf{b}), \mathsf{X}\neg\mathsf{XG}\neg\mathsf{p}(\mathtt{X})\} \subseteq \mathcal{F}(n)$. Here we have $\textsc{Red}(\mathsf{XXFp}(\mathsf{a})\mathsf{Pq}(\mathsf{b})) = \mathsf{XXFp}(\mathsf{a})\mathsf{Pq}(\mathsf{b})$ and $\textsc{Red}(\mathsf{X}\neg\mathsf{XG}\neg\mathsf{p}(\mathtt{X})) = \mathsf{XXFp}(\mathtt{X})$ which are unifiable with the mgu $\{\frac{\mathtt{X}}{\mathsf{a}}\}$. So n is closed.

Using this definition of closedness of nodes we can state the following criterion:

Termination by Closedness If n is a node such that n is closed. Then n can be skipped.

Here the term *skipping* means that it is not necessary to try to apply expansion and/or saturation rules to n. As we will see, closedness of a node n corresponds to unsatisfiability of $\mathcal{F}(n)$. Consequently we will have to apply rules in such a way that every path starting from the initial node leads to a closed node.

The termination criterion from above can be seen as a kind of *success criterion*, that is it enables a proof procedure to determine if the actual node has to be expanded or not. In contrast we can also state a dual (but much weaker) criterion of when the *complete* construction can be aborted. In order to state this criterion we need some more definitions.

Definition 5.3.4
Let $n = (\Phi, F, S)$ be a tableaux node. n is called *disjunctively expanded* if only the rules ($\mathsf{G1}$) and ($\mathsf{F2}$) can be applied to formulas from $\mathcal{F}(n)$ and no subformula of a formula in $\mathcal{F}(n)$ contains U or P.

So a node is disjunctively expanded if every possible continuation of the path from the root node leading to this node does not contain a node which contains a formula which can be split due to the presence of a disjunction symbol.

In addition to the property of being disjunctively expanded we have to state a property of a path which denotes the fact that every possible instantiation has indeed been carried out.

Definition 5.3.5
Let $\pi = (n_0, \ldots, n_k)$ be a path in a tableaux. Then π is called *completely instantiated* if for every ground term t occurring in $\mathcal{F}(n_0)$ and every node n_i on π such that there is a formula $\forall \mathtt{X} \varphi(\mathtt{X}) \in \mathcal{F}(n_i)$ there is a node n_j on π with $j \geq i$ and $\varphi(t) \in \mathcal{F}(n_j)$.

These two definitions enable us to state the following criterion for termination by failure.

Termination by Failure Let $\pi = (n_0, \ldots, n)$ be a completely instantiated path and let n be disjunctively expanded. Then the tableaux construction may be aborted if there are *no* formulas $\varphi_1, \varphi_2 \in \mathcal{F}(n)$ and no $i \geq 0$ such that at least one of the following conditions is fulfilled:

1. $\textsc{Red}(\varphi_1) = \mathsf{X}^i\psi_1$, $\textsc{Red}(\varphi_2) = \neg\mathsf{X}^i\psi_2$ and $\textsc{Red}(\psi_1)$ and $\textsc{Red}(\psi_2)$ are unifiable,

2. $\textsc{Red}(\varphi_1) = \neg\mathsf{X}^i\psi_1$, $\textsc{Red}(\varphi_2) = \mathsf{X}^j\mathsf{G}\psi_2$ for some $j \leq i$ and $\textsc{Red}(\psi_1)$ and $\textsc{Red}(\psi_2)$ are unifiable or

3. $\textsc{Red}(\varphi_1) = \neg\mathsf{X}^i\psi_1$, $\textsc{Red}(\varphi_2) = \neg\mathsf{X}^j\mathsf{F}\psi_2$ for some $j \leq i$ and $\textsc{Red}(\psi_1)$ and $\textsc{Red}(\psi_2)$ are unifiable.

This criterion is sound as the following lemma states.

Lemma 5.3.1

Let n be a node on an open path π in a such that the criterion *Termination by Failure* holds for n. Then $\mathcal{F}(n)$ is satisfiable.

Proof. Let $\mathcal{F}(n)$ be as required. Then every $\varphi \in \mathcal{F}(n)$ is either a first order literal or a FoLtl–literal containing at most the operators X, G and F (if at all). Define the set $\mathcal{J}_\pi$ as follows:

$$\mathcal{J}_\pi = \{\varphi \in \mathcal{F}(n) \mid \textsc{Var}(\varphi) = \emptyset, \textsc{Red}(\varphi) \text{ is positive }\}$$

Then $\mathcal{J}_\pi \models \mathcal{F}(n)$ and every temporally closed superset of $\mathcal{J}_\pi$ is a Herbrand–model of $\mathcal{F}(n)$. □

For the sake of simplicity we will introduce another formal concept. A node n is said to *fail* if the criterion *Termination by Failure* can be applied to the path π_n leading from the root node to n.

The Tableaux–Procedure

Having defined the rules of interest and the termination criteria, the definition of a proof procedure based on these rules is almost immediate. Assume that $P = \{P_1, \ldots, P_n\}$ is a Prolog(+T)–program and assume that $G = \psi_1 \wedge \cdots \wedge \psi_m$ is a Prolog(+T)–goal. As always assume that the P_i have the form $P_i = \varphi_i :\!-\psi_1^{(i)}, \ldots, \psi_{n_i}^{(i)}$. We construct the following formula $\varphi_{P,G}$ to be proven unsatisfiable:

$$\begin{aligned}
\varphi_{P,G} &= \left(\bigwedge_{i=1}^{n} P_i\right) \wedge :\!-G \\
&\equiv \left(\bigwedge_{i=1}^{n} \varphi_i :\!-\psi_1^{(i)}, \ldots, \psi_{n_i}^{(i)}\right) \wedge (:\!-\psi_1, \ldots, \psi_m) \\
&\equiv \left(\bigwedge_{i=1}^{n} \psi_1^{(i)} \wedge \cdots \wedge \psi_{n_i}^{(i)} \rightarrow \varphi_i\right) \wedge \left(\bigvee_{i=1}^{m} \neg\psi_i\right).
\end{aligned}$$

We then construct the initial tableaux node $n_0 = (\{\varphi_{P,G}\}, \emptyset, \emptyset)$ as described above and define two sets of nodes. The set N will consist of *all* nodes which have been constructed so far while the set U will contain all *unprocessed* nodes, that is nodes which have not yet been completely expanded. Initially we have $N = U = \{n_0\}$. We assume that U is realized as a queue, that is elements can only be taken from the *front* of U and put at the back of U. This ensures fairness of the *node selection* rule. See the discussion on page 89 for a treatment of the question of fairness regarding the selection rule for formulas. While U is not empty and no termination criterion is applicable we take the first node n_{act} from U and search for the first formula and the first rule which can be applied to this formula. New nodes are then created according to the definition of the rule which has been applied. We formalize this algorithm in Algorithm 1. There we will use the following notations: since U is assumed to be a queue structure, the only accessible element is the element at the front of U. This element will be returned by the function U.first. Consequently the operation U.pop will *remove* the first element from U. Insertion of elements is only possible at the *end* of U. So if $U = \{n_1, \ldots, n_k\}$ and $N_{\text{new}} = \{n_{\text{new}}\}$ (respectively $N_{\text{new}} = \{n_{\text{new}}^1, n_{\text{new}}^2\}$) is a set of newly created nodes, then $U \cup N_{\text{new}} = \{n_1, \ldots, n_k, n_{\text{new}}\}$ (respectively $U \cup N_{\text{new}} = \{n_1, \ldots, n_k, n_{\text{new}}^1, n_{\text{new}}^2\}$).

Algorithm 1 Tableaux algorithm for ground goals

Input:

- PROLOG(+T)–program P
- PROLOG(+T)–ground goal :–$G = G_1, \ldots, G_n$

Output: *yes* iff $P \models G_1 \wedge \cdots \wedge G_n$

```
1: construct $\varphi_{P,G}$
2: $n_0 \leftarrow (\{\varphi_{P,G}\}, \emptyset, \emptyset)$
3: $N \leftarrow \{n_0\}$, $U \leftarrow \{n_0\}$
Require: $U$ is realized as a queue
4: while $U \neq \emptyset$ do
5:   $n_{\text{act}} \leftarrow U$.first
6:   $U$.pop
7:   if $\mathcal{F}(n_{\text{act}})$ is not closed then
8:     if $n$ fails then
9:       return no
10:    else
11:      select a formula $\varphi \in \mathcal{F}(n_{\text{act}})$ and a rule $R$ applicable to $\varphi$
12:      apply $R$ to $\varphi$
13:      $N_{\text{new}} \leftarrow$ (set of) node(s) created by rule $R$
14:      $N \leftarrow N \cup N_{\text{new}}$
15:      $U \leftarrow U \cup N_{\text{new}}$
16:    end if
17:  end if
18: end while
19: return yes
```

5.3.2. Soundness and Completeness Issues

We will now address the topic of proving that the tableaux method described in the foregoing section is sound and refutation complete. The first part will be considered with soundness. Therefore we have to note that each node $n = (\mathcal{F}(n), F, S)$ in a tableaux can be seen as the root of a tableaux starting at this node.

The proofs from this section closely follow the proofs for the nontemporal tableaux procedure as presented in [20]. Our contribution is the treatment of the temporal constructs.

Lemma 5.3.2

Let n be any tableaux node which is the root of a closed tableaux. Then $F(n)$ is unsatisfiable.

Proof. Let h denote the height of the tableaux $\mathcal{T}$ rooted by n, that is h is the length of the longest path starting at n. We proceed by induction on h. First assume that $h = 0$. Then n is a leaf and since $\mathcal{T}$ is closed, we can distinguish the following three cases:

Case 1 There is a pair φ_1, φ_2 of literals in $\mathcal{F}(n)$ such that $\textsc{Red}(\varphi_1)$ and $\textsc{Red}(\neg\varphi_2)$ are unifiable. So clearly $\mathcal{F}(n)$ is unsatisfiable.

Case 2 There is $i \geq 0$ such that $\textsc{Red}(\varphi_1) = \mathsf{X}^i\psi_1^{(1)}\mathsf{P}\psi_2^{(1)}$, $\textsc{Red}(\varphi_2) = \mathsf{X}^i\psi^{(2)}$ and $\textsc{Red}(\psi_2^{(1)})$ and $\textsc{Red}(\psi^{(2)})$ are unifiable. Let $\sigma = \mathrm{mgu}(\textsc{Red}(\varphi_1), \textsc{Red}(\varphi_2))$ and $\psi = \sigma(\psi^{(2)}) = \sigma(\psi_2^{(1)})$ be given. Then we have

$$\begin{aligned}
& \sigma(\textsc{Red}(\varphi_1)) \wedge \sigma(\textsc{Red}(\varphi_2)) = \sigma(\mathsf{X}^i\psi_1^{(1)}\mathsf{P}\psi_2^{(1)}) \wedge \sigma(\mathsf{X}^i\psi^{(2)}) \\
= \;& \mathsf{X}^i\sigma(\psi_1^{(1)})\mathsf{P}\sigma(\psi_2^{(1)}) \wedge \mathsf{X}^i\sigma(\psi^{(2)}) \\
\equiv \;& \neg\mathsf{X}^i\sigma(\psi_2^{(1)}) \wedge (\mathsf{X}^i\sigma(\psi_1^{(1)}) \vee \mathsf{X}^{i+1}\sigma(\psi_1^{(1)})\mathsf{P}\sigma(\psi_2^{(1)})) \wedge \mathsf{X}^i\sigma(\psi^{(2)}) \\
\equiv \;& \neg\mathsf{X}^i\psi \wedge \mathsf{X}^i\psi \wedge (\mathsf{X}^i\sigma(\psi_1^{(1)}) \vee \mathsf{X}^{i+1}\sigma(\psi_1^{(1)})\mathsf{P}\sigma(\psi_2^{(1)})) \\
\equiv \;& \texttt{false} \wedge (\mathsf{X}^i\sigma(\psi_1^{(1)}) \vee \mathsf{X}^{i+1}\sigma(\psi_1^{(1)})\mathsf{P}\sigma(\psi_2^{(1)})) \\
\equiv \;& \texttt{false},
\end{aligned}$$

so clearly $\mathcal{F}(n)$ is unsatisfiable.

Case 3 There is $i \geq 0$ such that $\textsc{Red}(\varphi_1) = \neg\mathsf{X}^i\psi_1^{(1)}\mathsf{U}\psi_2^{(1)}$, $\textsc{Red}(\varphi_2) = \mathsf{X}^i\psi^{(2)}$ and $\textsc{Red}(\psi_2^{(1)})$ and $\textsc{Red}(\psi^{(2)})$ are unifiable. Again let $\sigma = \text{mgu}(\textsc{Red}(\varphi_1), \textsc{Red}(\varphi_2))$ and $\psi = \sigma(\psi^{(2)}) = \sigma(\psi_2^{(1)})$ be given. Then we have

$$\begin{aligned}
& \sigma(\textsc{Red}(\varphi_1)) \wedge \sigma(\textsc{Red}(\varphi_2)) = \sigma(\neg\mathsf{X}^i\psi_1^{(1)}\mathsf{U}\psi_2^{(1)}) \wedge \sigma(\mathsf{X}^i\psi^{(2)}) \\
= \; & \neg\mathsf{X}^i\sigma(\psi_1^{(1)})\mathsf{U}\sigma(\psi_2^{(1)}) \wedge \mathsf{X}^i\sigma(\psi^{(2)}) \\
\equiv \; & \neg\mathsf{X}^i\sigma(\psi_2^{(1)}) \wedge (\neg\mathsf{X}^i\sigma(\psi_1^{(1)}) \vee \neg\mathsf{X}^{i+1}\sigma(\psi_1^{(1)}\mathsf{U}\psi_2^{(1)})) \wedge \mathsf{X}^i\sigma(\psi^{(2)}) \\
\equiv \; & \neg\mathsf{X}^i\psi \wedge \mathsf{X}^i\psi \wedge (\neg\mathsf{X}^i\sigma(\psi_1^{(1)}) \vee \neg\mathsf{X}^{i+1}\sigma(\psi_1^{(1)}\mathsf{U}\psi_2^{(1)})) \\
\equiv \; & \mathtt{false} \wedge (\neg\mathsf{X}^i\sigma(\psi_1^{(1)}) \vee \neg\mathsf{X}^{i+1}\sigma(\psi_1^{(1)}\mathsf{U}\psi_2^{(1)})) \\
\equiv \; & \mathtt{false}.
\end{aligned}$$

So $\mathcal{F}(n)$ is unsatisfiable.

In every of the above cases we have shown that $\mathcal{F}(n)$ is unsatisfiable. So the case that $h = 0$ is proved.

Now assume that $h > 0$. Then a rule has been applied to n yielding one or two successor nodes n' and n''. Since $\mathcal{T}$ is closed, the tableaux $\mathcal{T}'$ and $\mathcal{T}''$ rooted by n' and n'' are also closed. Furthermore their height is $h-1$ and so by induction $\mathcal{F}(n')$ and $\mathcal{F}(n'')$ are both unsatisfiable. We distinguish the following cases of how n might have been expanded.

Case 1 Rule **($\wedge$ 1)** has been applied. Then $\mathcal{F}(n) = \{\varphi_1 \wedge \varphi_2\} \cup F'$ and $\mathcal{F}(n') = F' \cup \{\varphi_1, \varphi_2\}$. Since $\mathcal{F}(n')$ is unsatisfiable we have $\mathcal{J} \not\models F(n')$ for every $\mathcal{J}$. Fix one such $\mathcal{J}$. Since $\mathcal{J} \not\models F(n')$ at least one of the formulas in $\mathcal{F}(n')$ is not satisfied by $\mathcal{J}$. There are three possibilities:

Case 1.1 There is $\varphi_0 \in F'$ such that $\mathcal{J} \not\models \varphi_0$. This immediately gives $\mathcal{J} \not\models \mathcal{F}(n)$.

Case 1.2 $\mathcal{J} \not\models \varphi_1$. Then $\mathcal{J} \not\models \varphi_1 \wedge \varphi_2$ and so $\mathcal{J} \not\models \mathcal{F}(n)$.

Case 1.3 $\mathcal{J} \not\models \varphi_2$ This is analogous to the foregoing case.

So this proves that $\mathcal{F}(n)$ is unsatisfiable.

Case 2 Rule $(\vee\ \mathbf{1})$ has been applied. Then $\mathcal{F}(n) = \{\varphi_1 \vee \varphi_2\} \cup F'$ and $\mathcal{F}(n') = F' \cup \{\varphi_1\}$, $\mathcal{F}(n'') = F' \cup \{\varphi_2\}$. By induction both $\mathcal{F}(n')$ and $\mathcal{F}(n'')$ are unsatisfiable. Fix any $\mathcal{J}$. Then $\mathcal{J} \not\models \mathcal{F}(n')$ and $\mathcal{J} \not\models \mathcal{F}(n'')$. We again distinguish two cases:

Case 2.1 There is $\varphi_0 \in F'$ such that $\mathcal{J} \not\models \varphi_0$. This case is identical to case 1.1.

Case 2.2 $\mathcal{J} \models \varphi_0$ for every $\varphi_0 \in F'$. Then the unsatisfiability of $\mathcal{F}(n')$ and $\mathcal{F}(n'')$ gives $\mathcal{J} \not\models \varphi_1$ and $\mathcal{J} \not\models \varphi_2$. Therefore $\mathcal{J} \not\models \varphi_1 \vee \varphi_2$ and so $\mathcal{J} \not\models \mathcal{F}(n)$.

Case 3 Rule $(\neg\text{–}\mathbf{E})$ has been applied. Here $\mathcal{F}(n') = \{\neg\neg\varphi\} \cup F'$ and $\mathcal{F}(n') = F' \cup \{\varphi\}$. Since $\varphi \equiv \neg\neg\varphi$ we have $\mathcal{J} \models \varphi$ if and only if $\mathcal{J} \models \neg\neg\varphi$. Since $\mathcal{F}(n')$ is unsatisfiable $F(n)$ is also unsatisfiable.

Case 4 A rewrite rule or the reduction rule has been applied. Then the claim is proved exactly as in the foregoing case.

Case 5 One of the rules $(\forall\text{–}\mathbf{E})$ and $(\exists\text{–}\mathbf{E})$ has been applied. This case is trivial since if $\mathtt{X} \notin \textsc{Var}(\varphi)$ we have $\forall\mathtt{X}\varphi \equiv \exists\mathtt{X}\varphi \equiv \varphi$. So if n' is created from n by application of one of the above rules then $\mathcal{F}(n')$ is unsatisfiable if and only if $\mathcal{F}(n)$ is unsatisfiable.

Case 6 Rule $(\sigma\text{–}\mathbf{I_1})$ has been applied. Then $\mathcal{F}(n) = \{\forall\mathtt{X}\varphi(\mathtt{X})\} \cup F'$ and $\mathcal{F}(n') = F' \cup \{\varphi(t)\}$. Again fix any $\mathcal{J}$. Then $\mathcal{J} \not\models \mathcal{F}(n')$. The possible cases are:

Case 6.1 $\mathcal{J} \not\models \varphi_0$ for some $\varphi_0 \in F'$. Then we have the situation from case 1.1 and case 2.1.

Case 6.2 $\mathcal{J} \models \varphi_0$ for every $\varphi_0 \in F'$. Then $\mathcal{J} \not\models \varphi(t)$ that is $\mathcal{J} \not\models \sigma(\varphi(x))$ for $\sigma = \left\{\frac{\mathtt{X}}{t}\right\}$ and therefore $\mathcal{J} \not\models \forall\mathtt{X}\varphi(\mathtt{X})$. So $\mathcal{F}(n)$ is unsatisfiable.

Case 7 The application of the other rules are reduced to the foregoing cases by exploiting semantical definitions (for the temporal operators, the implication and the equivalence) and DeMorgan's laws (for the rules **(**$\wedge$ **2)**, **(**$\vee$ **2)**, **(**$\rightarrow$ **2)** and **(**$\leftrightarrow$ **2)**).

This proves the lemma. □

The above lemma especially holds for the starting node of a tableaux so we have the following easy corollary.

Corollary 5.3.1 (Soundness)

Let φ be any FoLtl-formula. If the tableaux for φ is closed, then φ is unsatisfiable.

Although rather technical, the proof that the tableaux construction is sound is quite straightforward exploiting only basic proof techniques. Proving refutation completeness is much more tricky. The tableaux construction technique known from first order logic uses so called *Hintikka–sets* (see [66]) in order to prove that for an unsatisfiable formula the tableaux construction is indeed capable of constructing a closed tableaux. Hintikka–sets are also of great use for proving the refutation completeness of the Prolog(+T) tableaux procedure as we will see now.

Intuitively a *Hintikka–set* is a set of formulas which is *semantically closed*, that if there is e.g. a formula $\varphi_1 \wedge \varphi_2$ in the set, then so are φ_1 and φ_2. The following definition formalizes this.

Definition 5.3.6 (Temporal Hintikka–Set)

Let P be a Prolog(+T)–program and let S be a set of FoLtl–formulas such that every function– or predicate–symbol occurring in S also occurs in P. Then S is called a *(temporal) Hintikka–set* if for every FoLtl–formula φ the following holds for every $i \geq 0$:

1. if $\varphi \in B_P^{\text{FoLtl}}$, then $\varphi \in S$ iff $\{\psi \mid \psi \equiv \neg\varphi\} \cap S = \emptyset$,

2. if $\varphi = \mathsf{X}^i(\varphi_1 \wedge \varphi_2)$, then $\varphi \in S$ implies $\mathsf{X}^i\varphi_1 \in S$ and $\mathsf{X}^i\varphi_2 \in S$,

3. if $\varphi = \mathsf{X}^i(\varphi_1 \vee \varphi_2)$, then $\varphi \in S$ implies $\mathsf{X}^i\varphi_1 \in S$ or $\mathsf{X}^i\varphi_2 \in S$,

4. if $\varphi = \mathsf{X}^i\forall\mathtt{X}\psi(\mathtt{X})$, then $\varphi \in S$ implies $\mathsf{X}^i\psi(t) \in S$ for every $t \in U_P$,

5. if $\varphi = \mathsf{X}^i\exists\mathtt{X}\psi(\mathtt{X})$, then $\varphi \in S$ implies $\mathsf{X}^i\psi(t) \in S$ for some ground term t,

6. if $\varphi = \mathsf{X}^i\mathsf{G}\psi$, then $\varphi \in S$ implies $\mathsf{X}^{i+j}\psi \in S$ for every $j \geq 0$,

7. if $\varphi = \mathsf{X}^i\mathsf{F}\psi$, then $\varphi \in S$ implies $\mathsf{X}^{i+j}\psi \in S$ for some $j \geq 0$,

8. if $\varphi = \neg\mathsf{X}^i\mathsf{G}\psi$, then $\varphi \in S$ implies there is $j \geq 0$ such that $\neg\mathsf{X}^{i+j}\psi \in S$ or $\neg\mathsf{X}^{i+j}\mathsf{G}\psi \in S$ for every $j \geq 0$,

9. if $\varphi = \neg\mathsf{X}^i\mathsf{F}\psi$, then $\varphi \in S$ implies $\neg\mathsf{X}^{i+j}\psi \in S$ for every $j \geq 0$,

10. if $\varphi = \mathsf{X}^i\varphi_1\mathsf{U}\varphi_2$, then $\varphi \in S$ implies $\mathsf{X}^i\varphi_2 \in S$ or $\mathsf{X}^i\varphi_1 \in S$ and $\mathsf{X}^{i+1}\varphi_1\mathsf{U}\varphi_2 \in S$,

11. if $\varphi = \varphi_1\mathsf{P}\varphi_2$, then $\varphi \in S$ implies $\neg\mathsf{X}^i\varphi_2 \in S$ and $\mathsf{X}^i\varphi_1 \in S$ or $\mathsf{X}^{i+1}\varphi_1\mathsf{P}\varphi_2 \in S$,

12. if $\varphi = \neg\mathsf{X}^i\varphi_1\mathsf{U}\varphi_2$, then $\varphi \in S$ implies $\neg\mathsf{X}^i\varphi_2 \in S$ and $\neg\mathsf{X}^i\varphi_1 \in S$ or $\neg\mathsf{X}^{i+1}\varphi_1\mathsf{U}\varphi_2 \in S$ and

13. if $\varphi = \neg\mathsf{X}^i\varphi_1\mathsf{P}\varphi_2$, then $\varphi \in S$ implies $\mathsf{X}^i\varphi_2 \in S$ or $\neg\mathsf{X}^i\varphi_1 \in S$ and $\neg\mathsf{X}^{i+1}\varphi_1\mathsf{P}\varphi_2 \in S$.

If S is a Hintikka–set constructed from symbols occurring in a program P, then we will also say that S is a *Hintikka–set with respect to (wrt.) P*. The definition of Hintikka sets can easily be adapted to deal with the special form of Prolog(+T)–rules by requiring $\varphi :- \psi_1, \ldots, \psi_n. \in S$ if and only if there is i such that $\psi_i \notin S$ or $\varphi \in S$.

The above definition is suitable for dealing with finite paths of tableaux nodes (as we will see soon). However since paths might also be of infinite length, we need some more definitions. In particular we need a concept of *maximality* of infinite paths. As before the formalisms used in the sequel closely follow [20].

Definition 5.3.7

Let $n = (\mathcal{F}(n), F, S)$ be a tableaux node, let $\varphi \in \mathcal{F}(n)$ be a formula and let $\mathbf{R}$ be a tableaux rule which can be applied to φ. Then $\mathbf{R}(\varphi)$ denotes the set of new formulas created by the application of $\mathbf{R}$.

Having introduced the operator $\mathbf{R}$ we can characterize paths as in tableaux to be maximal whether they are of finite length or of infinite length.

Definition 5.3.8

Let π be an open path. Then π is called *maximal* if

- π is finite and no more rules can be applied to $\mathcal{F}(\pi)$ or
- $\pi = (n_0, n_1, \ldots, n_i, \ldots)$ is infinite and for every $i \geq 0$, every $\varphi \in \mathcal{F}(n_i)$ and every rule $\mathbf{R}$ which can be applied to φ there is a node n_j $(j > i)$ such that $\mathbf{R}(\varphi) \subseteq \mathcal{F}(n_j)$.

Maximality of an open path will now turn out to be the key concept for proving refutation completeness.

Lemma 5.3.3

Let π be a maximal open path of tableaux nodes constructed from symbols occurring in a program P. Then $\mathcal{F}(\pi)$ is a (temporal) Hintikka–set wrt P.

Proof. First assume that π is finite. Since π is open and maximal, no rules can be applied to the formulas from $\mathcal{F}(\pi)$. In particular, no element of $\mathcal{F}(\pi)$ contains one of the operators G, F, U or P. Henceforth if we assume that $\mathcal{F}(\pi)$ is not a Hintikka–set, one of the following cases has to occur:

Case 1 There is a literal $\varphi \in \mathcal{F}(\pi)$ such that $\psi \in \mathcal{F}(\pi)$ for some ψ with $\psi \equiv \neg\psi$. But then π is closed which contradicts the assumptions on π.

Case 2 There is $\varphi = \varphi_1 \wedge \varphi_2 \in \mathcal{F}(\pi)$ such that $\varphi_1 \notin \mathcal{F}(\pi)$ or $\varphi_2 \notin \mathcal{F}(\pi)$. But in this case the rule $(\wedge\mathbf{1})$ can be applied to φ which contradicts the maximality of π.

Case 3 There is $\varphi = \varphi_1 \vee \varphi_2 \in \mathcal{F}(\pi)$ such that $\varphi_1 \notin \mathcal{F}(\pi)$ and $\varphi_2 \notin \mathcal{F}(\pi)$. In this case $(\vee\mathbf{1})$ can be applied which again contradicts the maximality of π.

Case 4 There is $\varphi = \forall \mathtt{X}\psi(\mathtt{X}) \in \mathcal{F}(\pi)$ and $t \in U_P$ such that $\psi(t) \notin \mathcal{F}(\pi)$. Then $(\sigma\text{–}\mathbf{I_1})$ can be applied to φ and therefore π is not maximal.

Case 5 There is $\varphi = \exists \mathtt{X}\psi(\mathtt{X}) \in \mathcal{F}(\pi)$ and $\psi(t) \notin \mathcal{F}(\pi)$ for every t. Then $(\sigma\text{–}\mathbf{I_2})$ can be applied to φ and therefore π is not maximal.

So in the case of a finite path π the claim is true. Now assume that π is of infinite length. Then we have more cases to distinguish. The first five cases are identical to the cases from above. So we will only have to consider the cases in which the operators G, F, U and P are involved. Assume that the assumptions from above are fulfilled, that is assume that π is maximal and open and assume that $\mathcal{F}(\pi)$ is not a Hintikka–set, i.e. assume that the conclusion of the implications from the definition of temporal Hintikka–sets are violated.

Case 6 There is $\mathsf{X}^i\mathsf{G}\psi \in \mathcal{F}(\pi)$ such that $\mathsf{X}^{i+1}\psi \notin \mathcal{F}(\pi)$ for some j. Then let j_0 be the minimal such j, that is $j_0 = \min\left\{j \mid \mathsf{X}^{i+j}\psi \notin \mathcal{F}(\pi)\right\}$. Since j_0 is minimal, we have $\mathsf{X}^{j_0-1}\psi \in \mathcal{F}(\pi)$ and $\mathsf{X}^{i+j_0-2}\mathsf{G}\psi \in \mathcal{F}(\pi)$. This implies $\mathsf{X}^{i+j_0-1}\mathsf{G}\psi \in \mathcal{F}(\pi)$ since π is maximal (otherwise $(\mathsf{G}\mathbf{1})$ could be applied). Again maximality now yields $\mathsf{X}^{i+j_0}\mathsf{G}\psi \in \mathcal{F}(\pi)$ and therefore $\mathsf{X}^{i+j_0}\psi \in \mathcal{F}(\pi)$ which is a contradiction.

Case 7 Let $\mathsf{X}^i\mathsf{F}\psi$ be in $\mathcal{F}(\pi)$.

a) Assume that $\mathsf{X}^{i+j}\psi \notin \mathcal{F}(\pi)$ for every $j \geq 0$. Since π is maximal, every possible application of the rule $(\vee\mathbf{1})$ has been carried out. Furthermore the path π corresponds to the path which contains the *right one* of the new formulas created by ($\vee\mathbf{1}$) (since instead there would be a minimal value j_0 such that $\mathsf{X}^{i+j_0}\psi \in \mathcal{F}(\pi)$). This yields $\mathsf{X}^{i+j}\mathsf{F}\psi \in \mathcal{F}(\pi)$ for every $j \geq 0$.

b) Now assume that there is $j \geq 0$ such that $\mathsf{X}^{i+j}\mathsf{F}\psi \notin \mathcal{F}(\pi)$. As in case 6 we chose the minimal value of all these j's, namely $j_0 = \min\{j \mid \mathsf{X}^{i+j}\mathsf{F}\psi \notin \mathcal{F}(\pi)\}$. So $\mathsf{X}^{i+j_0}\mathsf{F}\psi \notin \mathcal{F}(\pi)$ and $\mathsf{X}^{i+j_0-1}\mathsf{F}\psi \in \mathcal{F}(\pi)$. By maximality of π the application of ($\vee\mathbf{1}$) has been carried out and with $\mathsf{X}^{i+j_0}\mathsf{F}\psi \notin \mathcal{F}(\pi)$ we have $\mathsf{X}^{i+j_0-1}\psi \in \mathcal{F}(\pi)$.

Case 8 Let $\neg\mathsf{X}^i\mathsf{G}\psi$ be in $\mathcal{F}(\pi)$. We have the following cases:

a) for each $j \geq 0$ it holds that $\neg\mathsf{X}^{i+j}\psi \notin \mathcal{F}(\pi)$. Then in particular we have $\neg\mathsf{X}^i\psi \notin \mathcal{F}(\pi)$. But since π is maximal we have $\neg\mathsf{X}^{i+1}\mathsf{G}\psi \in \mathcal{F}(\pi)$ since otherwise ($\mathsf{G}\mathbf{2}$) could be applied and π is not maximal.

b) there is $j \geq 0$ such that $\neg\mathsf{X}^{i+j}\mathsf{G}\psi \notin \mathcal{F}(\pi)$. As before we chose the minimal value of all these j's: $j_0 = \min\{j \mid \neg\mathsf{X}^{i+j}\mathsf{G}\psi \notin \mathcal{F}(\pi)\}$. Then $\neg\mathsf{X}^{i+j_0}\mathsf{G}\psi \notin \mathcal{F}(\pi)$. Furthermore since $\neg\mathsf{X}^i\mathsf{G}\psi \in \mathcal{F}(\pi)$ we have $j_0 \geq 1$. Since j_0 is minimal we have $\neg\mathsf{X}^{i+j_0-1}\mathsf{G}\psi \in \mathcal{F}(\pi)$ and the maximality of π yields $\neg\mathsf{X}^{i+j_0-1}\psi \in \mathcal{F}(\pi)$.

Case 9 Let $\neg\mathsf{X}^i\mathsf{F}\psi$ be in $\mathcal{F}(\pi)$ and assume that there is $j \geq 0$ such that $\mathsf{X}^{i+j}\psi \in \mathcal{F}(\pi)$. Then we immediately have that there is a node n occurring on π which is closed. This contradicts the assumption that π is open.

Case 10 Let $\mathsf{X}^i\psi_1\mathsf{U}\psi_2$ be in $\mathcal{F}(\pi)$. If $\mathsf{X}^i\psi_2 \in \mathcal{F}(\pi)$ then the case is clear. Now assume that $\mathsf{X}^i\psi_2 \notin \mathcal{F}(\pi)$. Since π is maximal we have that the rule ($\mathsf{U}\mathbf{1}$) which is applicable has indeed been applied and creates the formula $\varphi' = \mathsf{X}^i\psi_1 \wedge \mathsf{X}^{i+1}\psi_1\mathsf{U}\psi_2$ to which ($\wedge\mathbf{1}$) can be applied. By maximality of π we have $\{\mathsf{X}^i\psi_1, \mathsf{X}^{i+1}\psi_1\mathsf{U}\psi_2\} \subseteq \mathcal{F}(\pi)$.

Case 11 Let $\mathsf{X}^i\psi_1\mathsf{P}\psi_2$ be in $\mathcal{F}(\pi)$. Then we immediately have $\neg\mathsf{X}^i\psi_2 \in \mathcal{F}(\pi)$ by maximality of π. Now if $\mathsf{X}^i\psi_1 \in \mathcal{F}(\pi)$, the case is clear. So assume that $\mathsf{X}^i\psi_1 \notin \mathcal{F}(\pi)$.

Then we have $\mathsf{X}^{i+1}\psi_1\mathsf{P}\psi_2 \in \mathcal{F}(\pi)$ since otherwise π is not maximal.

Case 12 Let $\neg\mathsf{X}^i\psi_1\mathsf{U}\psi_2$ be in $\mathcal{F}(\pi)$. By maximality of π we have $\{\neg\mathsf{X}^i\psi_2, \neg\mathsf{X}^i\psi_1 \vee \neg\mathsf{X}^{i+1}\psi_1\mathsf{U}\psi_2\} \subseteq \mathcal{F}(\pi)$ and therefore the application of $(\vee\mathbf{1})$ yields the desired result.

Case 13 Let $\neg\mathsf{X}^i\psi_1\mathsf{P}\psi_2$ be in $\mathcal{F}(\pi)$. By application of $(\mathsf{P}\mathbf{2})$ we have the following possibilities:

a) $\mathsf{X}^i\psi_2 \in \mathcal{F}(\pi)$. Then the case is clear.

b) $\mathsf{X}^i\psi_2 \notin \mathcal{F}(\pi)$. Then $(\mathsf{P}\mathbf{2})$ creates the formula $\neg\mathsf{X}^i\psi_1 \wedge \neg\mathsf{X}^{i+1}\psi_1\mathsf{P}\psi_2$ and an application of $(\wedge\mathbf{1})$ yields $\{\neg\mathsf{X}^i\psi_1, \neg\mathsf{X}^{i+1}\psi_1\mathsf{P}\psi_2\} \subseteq \mathcal{F}(\pi)$.

Therefore $\mathcal{F}(\pi)$ is a temporal Hintikka–set wrt. P and the lemma is proved. $\square$

As in first order logic, we are able to construct models for certain kinds of temporal Hintikka–sets.

Lemma 5.3.4

Let S be a temporal Hintikka–set such that $S = \mathcal{F}(\pi)$ for a maximal open path. Then S is satisfiable.

Proof. The claim is immediately by considering any interpretation which satisfies every ground *atom* from $\mathcal{F}(\pi)$. $\square$

Combining these two lemmas we have the following theorem.

Theorem 5.3.1 (Refutation–Completeness)

Let P be a PROLOG(+T)–program and let $G = \psi_1 \wedge \cdots \wedge \psi_n$. be a goal. If $P \models \psi_1 \wedge \cdots \wedge \psi_n$, then the tableaux rooted with $(\{\varphi_{P,G}\}, \emptyset, \emptyset)$ is closed.

Proof. We have $P \models \psi_1 \wedge \cdots \wedge \psi_n$ iff $P \cup \{G\} \models \Box$ iff $\varphi_{P,G}$ is unsatisfiable. Now assume that the tableaux rooted with $(\{\varphi_{P,G}\}, \emptyset, \emptyset)$ is not closed. Then there is a maximal open path π in this tableaux. We then have that $\mathcal{F}(\pi)$ is a temporal Hintikka–set wrt. P and consequently $\mathcal{F}(\pi)$ is satisfiable. This yields satisfiability of $\varphi_{P,G}$ which is a contradiction. So the theorem is proved. □

This result states the most desirable property of PROLOG(+T) and its inference mechanism. We have therefore shown that PROLOG(+T) is indeed an adequate programming language for the fragment of first order temporal logic under consideration. So we can proceed by treating the lattice properties of PROLOG(+T)–objects in order to justify our treatment of the refinement operations in the following chapters.

6. The Lattice Structure of Prolog(+T) objects

Contents

6.1. Subsumption **116**

6.2. The Lattice Structure of Literals **117**

6.2.1. Generalizations of Terms 117

6.2.2. Generalizations and Specializations of Literals 119

6.3. The Lattice Structure of Rules **125**

6.3.1. Greatest Specializations 125

6.3.2. Least Generalizations 129

We will now show how the concept of *subsumption* can be generalized from first order formulas to Prolog(+T)–objects. The main part of this generalization will be the integration of the temporal operators X, G, F, U and P, so the complicated part is section 6.2 where it will be shown that the lattice properties of the subsumption ordering carry over from first order logic literals to FoLtl–literals. In contrast the results from section 6.3 will be nearly identical to results from first order ILP.

6.1. Subsumption

During the construction of programs which satisfy a specification given by sets $\mathcal{E}^+ \subseteq B_P^{\text{FoLtl}}$ and $\mathcal{E}^- \subseteq B_P^{\text{FoLtl}}$ it might be necessary to specialize and/or generalize certain objects. In general, specialization and generalization should be related to the process of *largening* and *shrinking* the set of logical consequences (that is modifying a program P in order to yield a program P' which implies *more* – in the case of generalization – respectively *less* – in the case of specialization – than the original program P). However, the logical consequence relation $\models$ is undecidable and therefore one needs another ordering which is on the one hand decidable (that is it can be implemented on a computer) and on the other side closely related to logical consequence. Subsumption has turned out useful for this purpose (see [77] for a discussion of the difference between subsumption and implication in First Order Logic).

Informally, subsumption models the assumption that some object is *more general* than another one in the way that the more general object implies *more* than the less general one[1]. Formally, subsumption between literals is defined as follows.

Definition 6.1.1 (Subsumption for atoms, Plotkin [133])
Let φ_1 and φ_2 be literals from FoLtl. Then $\varphi_1 \succcurlyeq \varphi_2$ if and only if there is a substitution θ such that $\theta(\varphi_1) = \varphi_2$.

As one might expect, we will write $\varphi_1 \preccurlyeq \varphi_2$ if $\varphi_2 \succcurlyeq \varphi_1$, $\varphi_1 \succ \varphi_2$ if $\varphi_1 \succcurlyeq \varphi_2$ and not $\varphi_2 \succcurlyeq \varphi_1$, $\varphi_1 \prec \varphi_2$ if $\varphi_2 \succ \varphi_1$ and $\varphi_1 \approx \varphi_2$ if $\varphi_1 \succcurlyeq \varphi_2$ and $\varphi_2 \succcurlyeq \varphi_1$. Additionally we will write $\varphi_1 \not\succcurlyeq \varphi_2$ denoting that $\varphi_1 \succcurlyeq \varphi_2$ does not hold.

It is easily seen that $\succcurlyeq$ is reflexive and transitive. But it is not a partial ordering on the set of *all* FoLtl–literals since it is not anti–symmetric. Consider the literals $\varphi_1 = p(x)$

[1]This is due to the fact that the less general object can be constructed by instantiating the more general one.

and $\varphi_2 = p(y)$. Then for $\theta_1 = \left\{\frac{x}{y}\right\}$ and $\theta_2 = \left\{\frac{y}{x}\right\}$ we have $\theta_1(\varphi_1) = \varphi_2$ and $\theta_2(\varphi_2) = \varphi_1$ and therefore $\varphi_1 \succcurlyeq \varphi_2$ and $\varphi_2 \succcurlyeq \varphi_1$ but $\varphi_1 \neq \varphi_2$. However, φ_1 and φ_2 are variants.

We will use the ordering $\succcurlyeq$ in order to define a *quasi–order* $\succeq_s$ on literals which yields a lattice–structure. Therefore we define

- $\texttt{false} \succeq_s \varphi$ for every FoLtl–literal φ,
- $\varphi \succeq_s \texttt{true}$ for every FoLtl–literal φ and
- $\varphi_1 \succeq_s \varphi_2$ for $\varphi_1, \varphi_2 \notin \{\texttt{true}, \texttt{false}\}$ if and only if $\varphi_1 \succcurlyeq \varphi_2$.

The notations $\preceq_s$, $\succ_s$, $\prec_s$ and $\approx_s$ are defined as expected.

In the following section we will see how the ordering $\succeq_s$ yields a lattice structure in the set of all FoLtl–literals thus extending a well–known result from first order logic (see [133] and [134]).

6.2. The Lattice Structure of Literals

6.2.1. Generalizations of Terms

In order to present operations for computing least generalizations and greatest specializations of literals, we have to review some operations operating on terms. Recall that a *unifier* for two literals φ_1 and φ_2 is a substitution θ such that $\theta(\varphi_1) = \theta(\varphi_2)$. The process of *unification* can be reversed by constructing from φ_1 and φ_2 both a literal φ and substitutions θ_1 and θ_2 such that $\theta_1(\varphi) = \varphi_1$ and $\theta_2(\varphi) = \varphi_2$. Figure 6.1 illustrates the situation.

In [133] it has been shown that it is always possible to construct least generalizations and greatest specializations of terms. We will review the algorithm for constructing least generalizations here since we will need it later when computing least generalizations of literals.

$$\begin{tikzcd}
 & \varphi \arrow[dl, dash, "\theta_1"'] \arrow[dr, dash, "\theta_2"] & \\
\varphi_1 \arrow[dr, dash, "\theta"'] & & \varphi_2 \arrow[dl, dash, "\theta"] \\
 & \psi &
\end{tikzcd}$$

Figure 6.1.: Unification vs. Anti–unification

Algorithm 2 Anti–unification for terms

Input: terms t_1 and t_2

Output: term $\bar{t}$ such that $\bar{t}$ is a least generalization of t_1 and t_2

```
 1: $t'_1 \leftarrow t_1$, $t'_2 \leftarrow t_2$
 2: $\theta_1 \leftarrow \varepsilon$, $\theta_2 \leftarrow \varepsilon$
 3: $i \leftarrow 0$
Require: $\{z_i \mid i \in \mathbb{N}\}$ is a set of variables not occurring in $t_1$ and $t_2$
 4: if $t'_1 = t'_2$ then
 5:    return $\bar{t} = t'_1$
 6: else
 7:    $p \leftarrow$leftmost position at which $t'_1$ and $t'_2$ differ
 8:    $s \leftarrow t'_1|_p$
 9:    $t \leftarrow t'_2|_p$
10:    if there is $j \in \{1, \dots, i\}$ such that $\theta_1(z_j) = s$ and $\theta_2(z_j) = t$ then
11:       replace $t'_1|_p$ by $z_j$
12:       replace $t'_2|_p$ by $z_j$
13:    else
14:       $i \leftarrow i + 1$
15:       replace $t'_1|_p$ by $z_i$
16:       replace $t'_2|_p$ by $z_i$
17:       $\theta_1 = \theta_1 \circ \left\{\frac{z_i}{s}\right\}$
18:       $\theta_2 = \theta_2 \circ \left\{\frac{z_i}{t}\right\}$
19:       goto 4
20:    end if
21: end if
```

Algorithm 2 indeed produces a least generalization of the two input terms. This result can for example be found in [133] or [126]. Since this least generalization is a generalization with respect to subsumption we will denote the term returned by Algorithm 2 as

$$\mathrm{LGS}(t_1, t_2).$$

In later sections we will also denote any least generalization of literals φ_1 and φ_2 as $\mathrm{LGS}(\varphi_1, \varphi_2)$ and least generalizations of clauses C_1 and C_2 as $\mathrm{LGS}(C_1, C_2)$ since this will not cause any confusion.

6.2.2. Generalizations and Specializations of Literals

We will now apply the results from the last section in order to prove that the set of literals from FoLtl is a lattice ordered by the subsumption ordering $\succeq_s$. For formal reasons the set of all FoLtl–literals will from now on be denoted as $\mathcal{L}^{\text{FoLtl}}$. The proof will not be that difficult but rather long due to the different cases which have to be distinguished.

Theorem 6.2.1
$\left(\mathcal{L}^{\text{FoLtl}}, \succeq_s\right)$ is a lattice.

Proof. In order to prove the theorem we will show that both a least generalization and a greatest specialization of two given literals φ_1 and φ_2 exists in $\mathcal{L}^{\text{FoLtl}}$.

Specialization Let $\varphi_1, \varphi_2 \in \mathcal{L}^{\text{FoLtl}}$ be given. Assume without loss of generality that φ_1 and φ_2 have no variables in common, that is φ_1 and φ_2 are standardized apart.

1. if $\varphi_1 = \texttt{true}$ or $\varphi_2 = \texttt{true}$, then

$$\mathrm{GSS}(\varphi_1, \varphi_2) = \texttt{true},$$

2. a) if $\varphi_1 = \texttt{false}$, then

$$\mathrm{GSS}(\varphi_1, \varphi_2) = \varphi_2 \text{ and}$$

b) if $\varphi_2 = \texttt{false}$, then

$$\mathrm{GSS}(\varphi_1, \varphi_2) = \varphi_1,$$

3. if φ_1 and φ_2 are unifiable with $\sigma = \mathrm{mgu}(\varphi_1, \varphi_2)$, then

$$\mathrm{GSS}(\varphi_1, \varphi_2) = \sigma(\varphi_1) \text{ and}$$

4. if φ_1 and φ_2 are *not* unifiable, then

$$\mathrm{GSS}(\varphi_1, \varphi_2) = \texttt{true}.$$

It is obvious that $\mathrm{GSS}(\varphi_1, \varphi_2)$ is always a specialization of the original literals φ_1 and φ_2. It remains to show that it is indeed a greatest specialization. The only two nontrivial cases are that both $\varphi_1 \notin \{\texttt{true}, \texttt{false}\}$ and $\varphi_2 \notin \{\texttt{true}, \texttt{false}\}$. Consider the two cases:

Case 1 φ_1 and φ_2 are not unifiable. Then there cannot exist any literal ψ such that $\psi \neq \texttt{true}$ and $\varphi_1 \succeq_s \psi$ and $\varphi_2 \succeq_s \psi$ since then there would be substitutions θ_1 and θ_2 such that $\theta_1(\varphi_1) = \psi$ and $\theta_2(\varphi_2) = \psi$. Since φ_1 and φ_2 are standardized apart this would give

$$\begin{aligned} \theta_1(\varphi_1) &= (\theta_1 \circ \theta_2)(\varphi_1) = \psi \\ \theta_2(\varphi_2) &= (\theta_1 \circ \theta_2)(\varphi_2) = \psi \end{aligned}$$

So $\theta_1 \circ \theta_2$ is a unifier for φ_1 and φ_2 which is a contradiction to the assumption that φ_1 and φ_2 are not unifiable. So the claim is proved.

Case 2 φ_1 and φ_2 are unifiable. Then there is $\sigma = \mathrm{mgu}(\varphi_1, \varphi_2)$. if $\sigma(\varphi_1)$ is not a greatest specialization, then there is ψ and substitutions θ_1, θ_2 such that

$\theta_1(\varphi_1) = \psi$, $\theta_2(\varphi_2) = \psi$ and $\psi \succ_s \sigma(\varphi_1)$. But then there would be $\gamma \neq \varepsilon$ such that $\gamma(\psi) = \sigma(\varphi_1)$ and therefore σ would not be most general, which is a contradiction. So the claim is proved.

Generalization Let $\varphi_1, \varphi_2 \in \mathcal{L}^{\mathrm{FoLtl}}$ be given.

1. if φ_1 is positive and φ_2 is negative or if φ_1 is negative and φ_2 is positive, then $\mathrm{LGS}(\varphi_1, \varphi_2) := \mathtt{true}$.
2. if $\varphi_1 = \mathtt{false}$ or $\varphi_2 = \mathtt{false}$, then $\mathrm{LGS}(\varphi_1, \varphi_2) = \mathtt{false}$. Similarly if $\varphi_1 = \mathtt{true}$, then $\mathrm{LGS}(\varphi_1, \varphi_2) = \varphi_2$ and if $\varphi_2 = \mathtt{true}$, then $\mathrm{LGS}(\varphi_1, \varphi_2) = \varphi_1$.
3. if both φ_1 and φ_2 are negative, then assume that

$$\begin{aligned} \varphi_1 &= \neg\psi_1 \text{ and} \\ \varphi_2 &= \neg\psi_2 \end{aligned}$$

   and define

$$\mathrm{LGS}(\varphi_1, \varphi_2) = \neg\mathrm{LGS}(\psi_1, \psi_2)$$

4. if both φ_1 and φ_2 are positive, then
   a) if $\varphi_1 = p(t_1, \ldots, t_n)$ and $\varphi_2 = p(t'_1, \ldots, t'_n)$ for some p with $\alpha(p) = n$ and terms $t_1, \ldots, t_n, t'_1, \ldots, t'_n$, then

$$\mathrm{LGS}(\varphi_1, \varphi_2) = p(\mathrm{LGS}(t_1, t'_1), \ldots, \mathrm{LGS}(t_n, t'_n)),$$

   b) if $\varphi_1 = p(t_1, \ldots, t_n)$ and $\varphi_2 = q(t'_1, \ldots, t'_m)$ for p, q with $\alpha(p) = n$, $\alpha(q) = m$ and $t_1, \ldots, t_n, t'_1, \ldots, t'_m$ such that $p \neq q$, then

$$\mathrm{LGS}(\varphi_1, \varphi_2) = \mathtt{false},$$

c) if $\varphi_1 = \mathsf{X}\psi_1$ and $\varphi_2 = \mathsf{X}\psi_2$, then

$$\mathrm{LGS}(\varphi_1, \varphi_2) = \mathsf{X}\mathrm{LGS}(\psi_1, \psi_2),$$

d) if $\varphi_1 = \mathsf{G}\psi_1$ and $\varphi_2 = \mathsf{G}\psi_2$, then

$$\mathrm{LGS}(\varphi_1, \varphi_2) = \mathsf{G}\mathrm{LGS}(\psi_1, \psi_2),$$

e) if $\varphi_1 = \mathsf{F}\psi_1$ and $\varphi_2 = \mathsf{F}\psi_2$, then

$$\mathrm{LGS}(\varphi_1, \varphi_2) = \mathsf{F}\mathrm{LGS}(\psi_1, \psi_2),$$

f) if $\varphi_1 = \psi_1^{(1)}\mathsf{U}\psi_2^{(1)}$ and $\varphi_2 = \psi_1^{(2)}\mathsf{U}\psi_2^{(2)}$, then

$$\mathrm{LGS}(\varphi_1, \varphi_2) = \mathrm{LGS}(\psi_1^{(1)}, \psi_1^{(2)})\mathsf{U}\mathrm{LGS}(\psi_2^{(1)}, \psi_2^{(2)}),$$

g) if $\varphi_1 = \psi_1^{(1)}\mathsf{P}\psi_2^{(1)}$ and $\varphi_2 = \psi_1^{(2)}\mathsf{P}\psi_2^{(2)}$, then

$$\mathrm{LGS}(\varphi_1, \varphi_2) = \mathrm{LGS}(\psi_1^{(1)}, \psi_1^{(2)})\mathsf{P}\mathrm{LGS}(\psi_2^{(1)}, \psi_2^{(2)}) \text{ and}$$

h) in all other cases:

$$\mathrm{LGS}(\varphi_1, \varphi_2) = \mathtt{false}.$$

Again we will distinguish several cases.

Case 1 Case 1 from the above list occurs. Then the claim is trivial.

Case 2 Case 2 occurs. Then the claim is due to the definition of $\succeq_s$.

Case 3 We now proceed by induction on the structure of the literals. First assume that both φ_1 and φ_2 are nontemporal. Then the claim is due to results from [126] regarding the lattice structure of first order logic literals. Similarly we can prove the case in which both literals are negative by exploiting the assumption that the algorithm is correct for the subliterals under consideration. Now assume that φ_1 and φ_2 are of the form described in points c), d) and e). Then we can exploit the induction hypothesis for the literals ψ_1 and ψ_2 and the claim is immediate. Similarly we can treat the points f) and g). Finally in case h) no nontrivial least generalization can exist.

So the theorem is proved. □

The techniques from the proof of the above theorem are summarized in Algorithms 3 and 4.

Algorithm 3 Greatest Specialization of FoLtl–literals

Input: literals φ_1, φ_2

Output: $\mathrm{GSS}(\varphi_1, \varphi_2)$

```
1: if $\varphi_1$ and $\varphi_2$ are unifiable then
2:     $\sigma \leftarrow \mathrm{mgu}(\varphi_1, \varphi_2)$
3:     return $\sigma(\varphi_1)$
4: else
5:     return true
6: end if
```

Example 6.2.1

1. Consider $\varphi_1 = \mathsf{GF}p(\mathtt{X}, f(a))$ and $\varphi_2 = \mathsf{GF}q(a)$. Then $\mathrm{LGS}(\varphi_1, \varphi_2) = \mathtt{false}$ and $\mathrm{GSS}(\varphi_1, \varphi_2) = \mathtt{true}$.

2. Now consider $\varphi_1 = p(\mathtt{X}, \mathtt{X})$ and $p(f(a), b)$. Here we have $\mathrm{LGS}(\varphi_1, \varphi_2) = p(\mathtt{Z}_1, \mathtt{Z}_2)$ and $\mathrm{GSS}(\varphi_1, \varphi_2) = \mathtt{true}$.

3. Finally consider $\varphi_1 = \mathsf{GX}p(A, \mathtt{X})$ and $\varphi_2 = \mathsf{GX}p(\mathtt{Y}, b)$. Then $\mathrm{LGS}(\varphi_1) = \mathsf{GX}p(\mathtt{Z}_1, \mathtt{Z}_2)$ and $\mathrm{GSS}(\varphi_1, \varphi_2) = \mathsf{GX}p(a, b)$.

Algorithm 4 Least Generalization of FoLtl–literals

Input: literals φ_1, φ_2

Output: LGS(φ_1, φ_2)

```
1:  if $\varphi_1 = p(t_1, \dots, t_n)$ and $\varphi_2 = p(t'_1, \dots, t'_n)$ for some $p$ and $t_1, \dots, t_n, t'_1, \dots, t'_n$ then
2:      return $p(\mathrm{LGS}(t_1, t'_1), \dots, \mathrm{LGS}(t_n, t'_n))$
3:  end if
4:  if $\varphi_1 = \neg\psi_1$ and $\varphi_2 = \neg\psi_2$ then
5:      return $\neg\mathrm{LGS}(\psi_1, \psi_2)$
6:  end if
7:  if $\varphi_1 = \mathsf{X}\psi_1$ and $\varphi_2 = \mathsf{X}\psi_2$ then
8:      return $\mathsf{X}\mathrm{LGS}(\psi_1, \psi_2)$
9:  end if
10: if $\varphi_1 = \mathsf{G}\psi_1$ and $\varphi_2 = \mathsf{G}\psi_2$ then
11:     return $\mathsf{G}\mathrm{LGS}(\psi_1, \psi_2)$
12: end if
13: if $\varphi_1 = \mathsf{F}\psi_1$ and $\varphi_2 = \mathsf{F}\psi_2$ then
14:     return $\mathsf{F}\mathrm{LGS}(\psi_1, \psi_2)$
15: end if
16: if $\varphi_1 = \psi_1^{(1)}\mathsf{U}\psi_2^{(1)}$ and $\varphi_2 = \psi_1^{(2)}\mathsf{U}\psi_2^{(2)}$ then
17:     return $\mathrm{LGS}(\psi_1^{(1)}, \psi_1^{(2)})\mathsf{U}\mathrm{LGS}(\psi_2^{(1)}, \psi_2^{(2)})$
18: end if
19: if $\varphi_1 = \psi_1^{(1)}\mathsf{P}\psi_2^{(1)}$ and $\varphi_2 = \psi_1^{(2)}\mathsf{P}\psi_2^{(2)}$ then
20:     return $\mathrm{LGS}(\psi_1^{(1)}, \psi_1^{(2)})\mathsf{P}\mathrm{LGS}(\psi_2^{(1)}, \psi_2^{(2)})$
21: end if
22: return false
```

6.3. The Lattice Structure of Rules

Algorithms 3 and 4 from the last section can now be used in order to compute least generalizations and greatest specializations of clauses, that is least generalizations and greatest specializations of PROLOG(+T)–rules. For this purpose we will adapt a proof for the existence of least generalizations and greatest specializations of first order logic clauses which might for example be found in [126]. The subsumption ordering for rules is defined as follows.

Definition 6.3.1 (Subsumption for Rules, Plotkin [133])
Let C_1 and C_2 be PROLOG(+T)–rules (represented as sets of literals). Then $C_1 \succeq_s C_2$ if and only if there is a substitution θ such that $\theta(C_1) \subseteq C_2$.

The symbols $\succ_s$, $\preceq_s$ and $\prec_s$ are then defined as usual.

6.3.1. Greatest Specializations

As in the last section, the simpler part is the computation of greatest specializations of clauses. Therefore we will adapt a technique described in [126]. We will see that the greatest specialization of two PROLOG(+T)–rules is in general not unique. This is due to the fact that we allow negated atoms in the tails of rules.

Assume that

$$\begin{aligned} C_1 &= \varphi_1 :- \psi_1^{(1)}, \ldots, \psi_{n_1}^{(1)} \text{ and} \\ C_2 &= \varphi_2 :- \psi_1^{(2)}, \ldots, \psi_{n_2}^{(2)} \end{aligned}$$

are given. We identify C_1 and C_2 with the sets of literals involved in these rules, that is we concentrate on

$$\begin{aligned} S_{C_1} &= \left\{\neg\psi_1^{(1)}, \dots, \neg\psi_{n_1}^{(1)}, \varphi_1\right\} \text{ and} \\ S_{C_2} &= \left\{\neg\psi_1^{(2)}, \dots, \neg\psi_{n_2}^{(2)}, \varphi_2\right\} \end{aligned}$$

as the objects of interest. We now build the set $C = S_{C_1} \cup S_{C_2}$ and construct a set of PROLOG(+T)–rules from this set each of which is a greatest specialization of C_1 and C_2. But in order to do this we define a helper function TAIL which will be useful in the definition.

Let $T = \{\gamma_1, \dots, \gamma_k\}$ be a set of FOLTL–literals. Then the string TAIL(T) is defined as

$$\text{TAIL}(T) = \neg\gamma_1, \dots, \neg\gamma_k,$$

where $\neg\neg\gamma_i$ will be identified with γ_i for $i = 1, \dots, k$.

Now assume that $S = \{l_1, \dots, l_{n_1+n_2+2}\}$. We then build the set L consisting of all PROLOG(+T)–rules which can be constructed using the following scheme: if l_i is positive, then L contains the rule

$$l_i \text{ :– TAIL}(S \setminus \{l_i\}).$$

Algorithm 5 illustrates this scheme.

The properties of Algorithm 5 are summarized in the following theorem.

Theorem 6.3.1

Let C_1 and C_2 be PROLOG(+T)–rules. Then Algorithm 5 computes a set of rules each of which is a greatest specialization of C_1 and C_2.

Proof. Let C_1 and C_2 be given. Without loss of generality we can assume that C_1 and C_2 are standardized apart. Let C be any rule computed by Algorithm 5. Then for $S = S_{C_1} \cup S_{C_2}$ we have $S_{C_1} \subseteq S$ and $S_{C_2} \subseteq S$ so both $C_1 \succeq_s C$ and $C_2 \succeq_s C$ holds, that is C is a specialization under subsumption of C_1 and C_2. Now consider any rule

Algorithm 5 Greatest Specialization of PROLOG(+T)–rules

Input: PROLOG(+T)–rules C_1, C_2
Output: set of greatest specializations of C_1, C_2

```
Require: $C_1 = \varphi_1 :- \psi_1^{(1)}, \ldots, \psi_{n_1}^{(1)}$, $C_2 = \varphi_1 :- \psi_1^{(2)}, \ldots, \psi_{n_2}^{(2)}$
 1: $L \leftarrow \emptyset$
 2: $S_{C_1} \leftarrow \left\{\neg\psi_1^{(1)}, \ldots, \neg\psi_{n_1}^{(1)}, \varphi_1\right\}$
 3: $S_{C_2} \leftarrow \left\{\neg\psi_1^{(2)}, \ldots, \neg\psi_{n_2}^{(2)}, \varphi_2\right\}$
 4: $S \leftarrow S_{C_1} \cup S_{S_2}$
Require: $S = \{l_1, \ldots, l_o\}$
 5: for $i = 1, \ldots, o$ do
 6:     if $l_i$ is positive then
 7:         $L \leftarrow L \cup \{l_i :- \text{TAIL}(S \setminus \{l_i\})\}$
 8:     end if
 9: end for
10: return $L$
```

$\bar{C}$ such that $C_1 \succeq_s \bar{C}$ and $C_2 \succeq_s \bar{C}$. Then there are substitutions θ_1 and θ_2 such that $\theta_1(C_1) \subseteq \bar{C}$ and $\theta_2(C_2) \subseteq \bar{C}$ and θ_1 and θ_2 only replace variables in C_1 and C_2 (since C_1 and C_2 are standardized apart). Define $\theta = \theta_1 \cup \theta_2$. Then

$$\begin{aligned}
\theta\left(S_C\right) &= \theta\left(S_{C_1} \cup S_{C_2}\right) \\
&= \theta\left(S_{C_1}\right) \cup \theta\left(S_{C_2}\right) \\
&= \theta_1\left(S_{C_1}\right) \cup \theta_2\left(S_{C_2}\right) \\
&\subseteq S_{\bar{C}}.
\end{aligned}$$

So $C \succeq_s \bar{C}$. Since C was chosen arbitrary from the set of rules computed by Algorithm 5 the claim is proved. □

Although the syntactical form of a greatest specialization of PROLOG(+T)–rules is not uniquely determined we can adapt the notation GSS(C_1, C_2). This is justified since for every $C^{(1)}, C^{(2)}$ computed by Algorithm 5 given the inputs C_1 and C_2 we have $S_{C^{(1)}} =$

$S_{C^{(2)}} = S_{C_1} \cup S_{C_2}$ and therefore $C^{(1)} \equiv C^{(2)}$. Consequently

$$\mathrm{GSS}(C_1, C_2)$$

will from now on be used in order to denote *any* of the rules computed by Algorithm 5.

We will now close this section by presenting a simple example of how Algorithm 5 creates greatest specializations of PROLOG(+T)–rules.

Example 6.3.1

Consider the rules

$$\begin{aligned} C_1 &= \mathsf{Gp(a)} :\!- \mathsf{XXp}(\mathtt{X}_1).\text{ and} \\ C_2 &= \mathsf{XFp}(\mathtt{X}_2) :\!- \mathtt{not}(\mathtt{r}(\mathtt{X}_2)), \mathsf{Gp(b)}. \end{aligned}$$

Then we have

$$\begin{aligned} S_{C_1} &= \{\mathtt{not}(\mathsf{XXp}(\mathtt{X}_1)), \mathsf{Gp(a)}\}\ \text{ and} \\ S_{C_2} &= \{\mathtt{r}(\mathtt{X}_2), \mathtt{not}(\mathsf{Gp(b)}), \mathsf{XFq}(\mathtt{X}_2)\} \end{aligned}$$

and therefore

$$S = \{\mathtt{not}(\mathsf{XXp}(\mathtt{X})), \mathsf{Gp(a)}, \mathtt{r}(\mathtt{X}), \mathtt{not}(\mathsf{Gp(b)}), \mathsf{XFq}(\mathtt{X})\}\,.$$

The rules generated by Algorithm 3 are the following:

1. $\mathsf{Gp(a)} :\!- \mathsf{XXp}(\mathtt{X}_1), \mathtt{not}(\mathtt{r}(\mathtt{X}_2)), \mathtt{not}(\mathsf{Gp(b)}), \mathtt{not}(\mathsf{XFq}(\mathtt{X}_2))$.,

2. $\mathtt{r}(\mathtt{X}_2) :\!- \mathsf{XXp}(\mathtt{X}_1), \mathtt{not}(\mathsf{Gp(a)}), \mathsf{Gp(b)}, \mathtt{not}(\mathsf{XFq}(\mathtt{X}_2))$. and

3. $\mathsf{XFq}(\mathtt{X}_2) :\!- \mathsf{XXp}(\mathtt{X}_1), \mathtt{not}(\mathsf{Gp(a)}), \mathtt{not}(\mathtt{r}(\mathtt{X}_2)), \mathsf{Gp(b)}$.

6.3.2. Least Generalizations

What remains to be established is the existence of least generalizations of PROLOG(+T)–rules under subsumption. For first order literals the concept of *compatibility* has been used for the proof of the existence of least generalizations of clauses. Two first order literals φ_1 and φ_2 are considered *compatible* if they are either both positive or both negative and if they start with the same predicate symbol. For FOLTL–literals the situation is slightly more complicated. Intuitively we should consider literals compatible if they are either both positive or both negative and if they contain the same temporal operators in the same order. Formally we will present a technique which constructs for a literal φ a tree TREE(φ) from which a tuple TEMP(φ) of *words* built up from the operators involved in the literal φ can be extracted. Two literals φ_1 and φ_2 are then considered *compatible* if they yield identical sets TEMP(φ_1) and TEMP(φ_2).

Given a literal φ we will now show how to construct a *labeled graph* TREE(φ). Recall from chapter 5.2 that a *labeled graph* is a tuple $T = (V, E, l)$ consisting of a finite set V containing the *vertices* or *nodes*, a set $E \subseteq V \times V$ containing the edges and a mapping l. Here l has the form $l : V \to \{\mathsf{U}, \mathsf{P}, \mathsf{X}, \mathsf{G}, \mathsf{F}, \neg\} \cup P \cup \mathcal{T}$. We will partition the set V into three sets V_t, V_p and V_f containing so called *temporal nodes*, *predicate nodes* and *function nodes*, that is $V = V_t \dot\cup V_P \dot\cup V_f$. The construction of TREE(φ) is now given by induction on the form of φ.

Case 1 $\varphi = \mathsf{p}(t_1, \ldots, t_n) \in B_P$ is a first order atom. Then we set

$$\text{TREE}(\varphi) = (\{v_0, v_1, \ldots, v_n\}, \{(v_0, v_i) \mid i = 1, \ldots, n\}, l)\,,$$

where l is defined by

$$\begin{aligned} l(v_0) &= \mathsf{p} \text{ and} \\ l(v_i) &= t_i \text{ for } i > 0 \end{aligned}$$

and

$$\begin{aligned} V_t &= \emptyset, \\ V_f &= \{v_1, \ldots, v_n\} \text{ and} \\ V_p &= \{v_0\}. \end{aligned}$$

Case 2 $\varphi = \texttt{not}(\psi)$ for some PROLOG(+T)–literal ψ. Then assume that $\textsc{Tree}(\psi) = \left(\bar{V}, \bar{E}, \bar{l}\right)$ is given. If $\bar{V} = \left\{\bar{v}_1, \ldots, \bar{v}_{|\bar{V}|}\right\} = \bar{V}_t \mathbin{\dot{\cup}} \bar{V}_p \mathbin{\dot{\cup}} \bar{V}_f$ then we define

$$\begin{aligned} V &= \left\{v_0, \bar{v}_1, \ldots, \bar{v}_{|\bar{V}|}\right\} \text{ for some } v_0 \notin \bar{V}, \\ V_t &= \bar{V}_t, \\ V_f &= \bar{V}_f, \\ V_p &= \bar{V}_p \cup \{v_0\}, \\ E &= \bar{E} \cup \left\{(v_0, \bar{v}) \mid \bar{v} \in \bar{V} \text{ such that } (\bar{\bar{v}}, \bar{v}) \notin \bar{E} \text{ for each } \bar{\bar{v}} \in \bar{V}\right\}, \\ l(v_0) &= \texttt{not} \text{ and} \\ l(v) &= \bar{l}(v) \text{ for each } v \neq v_0. \end{aligned}$$

Case 3 $\varphi = \oplus\psi$ for some PROLOG(+T)–literal ψ and some $\oplus \in \{\mathsf{X}, \mathsf{G}, \mathsf{F}\}$. Then assume that $\textsc{Tree}(\psi) = \left(\bar{V}, \bar{E}, \bar{l}\right)$ is given. If $\bar{V} = \left\{\bar{v}_1, \ldots, \bar{v}_{|\bar{V}|}\right\} = \bar{V}_t \mathbin{\dot{\cup}} \bar{V}_p \mathbin{\dot{\cup}} \bar{V}_f$, then define

$$\begin{aligned} V &= \left\{v_0, \bar{v}_1, \ldots, \bar{v}_{|\bar{V}|}\right\} \text{ for some } v_0 \notin \bar{V}, \\ V_t &= \bar{V}_t, \\ V_f &= \bar{V}_f, \\ V_p &= \bar{V}_p \cup \{v_0\}, \\ E &= \bar{E} \cup \left\{(v_0, \bar{v}) \mid \bar{v} \in \bar{V} \text{ such that } (\bar{\bar{v}}, \bar{v}) \notin \bar{E} \text{ for each } \bar{\bar{v}} \in \bar{V}\right\}, \end{aligned}$$

$$\begin{aligned} l(v_0) &= \oplus \text{ and} \\ l(v) &= \bar{l}(v) \text{ for each } v \neq v_0. \end{aligned}$$

Case 4 $\varphi = \psi_1 \oplus \psi_2$ for PROLOG(+T)–literals ψ_1 and ψ_2 and some $\oplus \in \{\mathsf{U}, \mathsf{P}\}$. Assume that $\text{TREE}(\psi_1) = (\bar{V}_1, \bar{E}_1, \bar{l}_1)$ with $\bar{V}_1 = \bar{V}_{1,t} \,\dot{\cup}\, \bar{V}_{1,p} \,\dot{\cup}\, \bar{V}_{1,f}$ and $\text{TREE}(\psi_2) = (\bar{V}_2, \bar{E}_2, \bar{l}_2)$ with $\bar{V}_2 = \bar{V}_{2,t} \,\dot{\cup}\, \bar{V}_{2,p} \,\dot{\cup}\, \bar{V}_{2,f}$ are given such that $\bar{V}_1 \cap \bar{V}_2 = \emptyset$. Choose some new $v_0 \notin \bar{V}_1 \cup \bar{V}_2$ and set

$$\begin{aligned} V &= \{v_0\} \cup \bar{V}_1 \cup \bar{V}_2, \\ V_t &= \bar{V}_{1,t} \cup \bar{V}_{2,t} \cup \{v_0\}, \\ V_p &= \bar{V}_{1,p} \cup \bar{V}_{2,p}, \\ V_f &= \bar{V}_{1,f} \cup \bar{V}_{2,f}, \\ E &= \bar{E}_1 \cup \bar{E}_2 \cup \left\{ (v_0, v) \mid \begin{array}{c} v \in \bar{V}_1 \cup \bar{V}_2 \text{ such that } (\bar{v}, v) \notin \bar{E}_1 \cup \bar{E}_1 \\ \text{for each } \bar{v} \in \bar{V}_1 \cup \bar{V}_2 \end{array} \right\}, \\ l(v_0) &= \oplus, \\ l(v) &= \bar{l}_1(v) \text{ for } v \in \bar{V}_1 \text{ and} \\ l(v) &= \bar{l}_2(v) \text{ for } v \in \bar{V}_2. \end{aligned}$$

Example 6.3.2

1. Assume that $\varphi_1 = \mathsf{q(a)Up(b)}$ and $\varphi_2 = \mathsf{q(c)Up(a)}$. The graphs $\text{TREE}(\varphi_1)$ and $\text{TREE}(\varphi_2)$ are depicted in Figures 6.2 and 6.3 where the nodes from V_f are drawn as squares while all other nodes are drawn as circles.

2. Now assume that $\varphi_3 = \mathsf{q(c)UXp(a)}$. Then $\text{TREE}(\varphi_3)$ is as depicted in Figure 6.4.

Now recall that a *path* from a node v_1 to a node v_2 in a (labeled) graph $G = (V, E, l)$ is defined as follows:

1. either $(v_1, v_2) \in E$ or

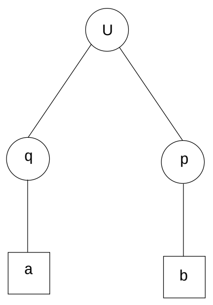

Figure 6.2.: TREE(`q(a)Up(b)`)

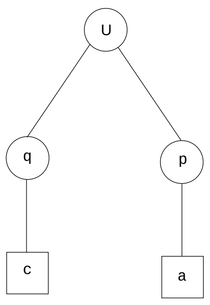

Figure 6.3.: TREE(`q(c)Up(a)`)

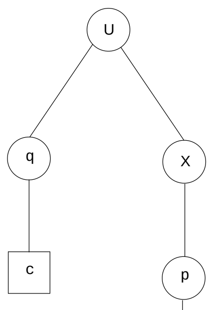

Figure 6.4.: TREE(`q(c)UXp(a)`)

2. there is $v \in V$ such that $(v_1, v) \in E$ and there is a path from v to v_2 in G.

Paths can be described very naturally by giving the sequence of nodes on this path. We will from now on describe paths as $\pi = (v_1, \ldots, v_n)$ and call π a path of *length n*. Having constructed $\textsc{Tree}(\varphi)$ from the literal φ, we can extract the information $\textsc{Temp}(\varphi)$ as follows: assume that the nodes from V_f are numbered in ascending order from *left to right.* If $V_f = \{v_1, \ldots, v_k\}$ then $\textsc{Temp}(\varphi) = (s_1, \ldots, s_k)$ where s_i is defined as follows:

1. Let $\pi = (v_0, \ldots, v_{k_i}, v_i)$ be the uniquely determined path from v_0 to v_i and

2. $s_i = l(v_0) \circ \cdots \circ l(v_{k_i})$ where $\circ$ denotes the concatenation of words.

Example 6.3.3

Again consider the literals from Example 6.3.2. We then have

$$\begin{aligned} \textsc{Temp}(\varphi_1) &= (\mathsf{Uq}, \mathsf{Up})\,, \\ \textsc{Temp}(\varphi_2) &= (\mathsf{Uq}, \mathsf{Up}) \text{ and} \\ \textsc{Temp}(\varphi_3) &= (\mathsf{Uq}, \mathsf{UXp})\,. \end{aligned}$$

The construction of $\textsc{Temp}(\varphi)$ from a given literal φ allows the extension of the concept of *compatibility* from first order logic.

Definition 6.3.2

Let φ_1 and φ_2 be $\textsc{Prolog}(+\mathrm{T})$–literals. φ_1 and φ_2 are called *compatible* if $\textsc{Temp}(\varphi_1) = \textsc{Temp}(\varphi_2)$.

Intuitively φ_1 and φ_2 are assumed to be compatible if they only differ in their subterms. So compatibility is a criterion for the existence of a nontrivial least generalization of two literals.

Example 6.3.4

Again consider the literals φ_1, φ_2 and φ_3 from Example 6.3.2. From Example 6.3.3 it is clear that $\text{TEMP}(\varphi_1) = \text{TEMP}(\varphi_2) = (\mathsf{Uq}, \mathsf{Up})$. So φ_1 and φ_2 are compatible. On the other hand $\text{TEMP}(\varphi_3) \neq \text{TEMP}(\varphi_1)$. So φ_1 and φ_3 are not compatible.

The concept of compatibility of literals will now be used in order to compute least generalizations of PROLOG(+T)–rules. Essentially the procedure is identical to a procedure which is known from first order ILP. Assume that

$$\begin{aligned} C_1 &= \varphi_1 :- \psi_1^{(1)}, \ldots, \psi_{n_1}^{(1)}. \text{ and} \\ C_2 &= \varphi_2 :- \psi_1^{(2)}, \ldots, \psi_{n_2}^{(2)}. \end{aligned}$$

are two PROLOG(+T)–rules. We will again work with the sets S_{C_1} and S_{C_2} of literals which represent these rules. So assume that

$$\begin{aligned} S_{C_1} &= \left\{\neg\psi_1^{(1)}, \ldots, \neg\psi_{n_1}^{(1)}, \varphi_1\right\} = \left\{\chi_0^{(1)}, \chi_1^{(1)}, \ldots, \chi_{n_1}^{(1)}\right\} \text{ and} \\ S_{C_2} &= \left\{\neg\psi_1^{(2)}, \ldots, \neg\psi_{n_2}^{(2)}, \varphi_2\right\} = \left\{\chi_0^{(2)}, \chi_1^{(2)}, \ldots, \chi_{n_2}^{(2)}\right\}, \end{aligned}$$

where $\chi_0^{(i)} = \varphi_i$ and $\chi_j^{(i)} = \neg\psi_j^{(i)}$ for $i = 1, 2$ and $j > 0$. Let

$$Sel = \left\{\left(\chi_{i_0}^{(1)}, \chi_{j_0}^{(2)}\right), \ldots, \left(\chi_{i_k}^{(1)}, \chi_{j_k}^{(2)}\right)\right\}$$

be the set of all pairs of compatible literals from S_{C_1} and S_{C_2}. If $Sel = \emptyset$ or if Sel does not contain at least one pair of *positive* literals, then we define

$$\text{LGS}(C_1, C_2) = \{\mathtt{false}\}.$$

This is reasonable since

1. if there are no compatible literals, then no rule C and no pair of substitutions

$\sigma_1, \sigma_2 \neq \varepsilon$ such that $\sigma(S_C) \subseteq S_{C_1}$ and $\sigma_2(S_C) \subseteq S_{C_2}$ can exist (assuming that $S_C \neq \emptyset$) and

2. if there is no pair of compatible atoms, then there is no literal which can be generalized in order to yield the *head* of the generalized rule.

We will now concentrate on the case that $Sel \neq \emptyset$, that is there is at least one pair of compatible literals. In order to compute the set of least generalizations of C_1 and C_2 we first adapt a technique presented in [133] which transforms literals to terms. Therefore assume that $\mathrm{sig} = (\mathcal{X}, F, P, \alpha)$ is the signature from which C_1 and C_2 are built. We extend this signature to $\mathrm{sig}_{\mathrm{ext}} = (\mathcal{X}, F_{\mathrm{ext}}, P_{\mathrm{ext}}, \alpha_{\mathrm{ext}})$ as follows:

- $F_{\mathrm{ext}} = F \cup \left\{\mathtt{f}_{p_1}, \ldots, \mathtt{f}_{p_{|P|}}\right\} \cup \{\mathtt{f}_{\mathrm{not}}\} \cup \{\mathtt{f}_{\mathrm{next}}, \mathtt{f}_{\mathrm{always}}, \mathtt{f}_{\mathrm{finally}}, \mathtt{f}_{\mathrm{until}}, \mathtt{f}_{\mathrm{precedes}}\}$ assuming that $P = \left\{\mathtt{p}_1, \ldots, \mathtt{p}_{|P|}\right\}$,
- $P_{\mathrm{ext}} = P \cup \{\mathtt{p}_{\mathrm{new}}\}$ for some symbol $\mathtt{p}_{\mathrm{new}} \notin P$ and
-

$$\begin{aligned}
\alpha_{\mathrm{ext}}(\mathtt{f}) &= \alpha(\mathtt{f}) \text{ for } \mathtt{f} \in F, \\
\alpha_{\mathrm{ext}}\left(\mathtt{f}_{\mathtt{p}_i}\right) &= \alpha(\mathtt{p}_i) \text{ for } i = 1, \ldots, |P|, \\
\alpha_{\mathrm{ext}}(\mathtt{p}_{\mathrm{new}}) &= k, \\
\alpha_{\mathrm{ext}}(\mathtt{f}_{\mathrm{not}}) &= \alpha_{\mathrm{ext}}(\mathtt{f}_{\mathrm{next}}) = \alpha_{\mathrm{ext}}(\mathtt{f}_{\mathrm{always}}) = \alpha_{\mathrm{ext}}(\mathtt{f}_{\mathrm{finally}}) = 1 \text{ and} \\
\alpha_{\mathrm{ext}}(\mathtt{f}_{\mathrm{until}}) &= \alpha_{\mathrm{ext}}(\mathtt{f}_{\mathrm{precedes}}) = 2.
\end{aligned}$$

Using the signature $\mathrm{sig}_{\mathrm{ext}}$ we define an operation $\textsc{Term} : \mathcal{L}_t(\mathrm{sig}) \to \mathcal{T}(\mathrm{sig}_{\mathrm{ext}})$ which maps literals to terms. Term is defined inductively as follows:

- if $\varphi = \mathtt{p}(t_1, \ldots, t_n)$ for some symbol $\mathtt{p} \in P$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathrm{sig})$, then $\textsc{Term}(\varphi) = \mathtt{f}_{\mathtt{p}}(t_1, \ldots, t_n)$,
- if $\varphi = \mathtt{not}(\psi)$ for some literal $\psi \in \mathcal{L}_t(\mathrm{sig})$, then $\textsc{Term}(\varphi) = \mathtt{f}_{\mathrm{not}}(\textsc{Term}(\psi))$,

- – if $\varphi = \mathsf{X}\psi$ for some $\psi \in \mathcal{L}_t(\text{sig})$, then $\textsc{Term}(\varphi) = \mathtt{f}_{\text{next}}(\textsc{Term}(\psi))$,
  – if $\varphi = \mathsf{G}\psi$ for some $\psi \in \mathcal{L}_t(\text{sig})$, then $\textsc{Term}(\varphi) = \mathtt{f}_{\text{always}}(\textsc{Term}(\psi))$ and
  – if $\varphi = \mathsf{F}\psi$ for some $\psi \in \mathcal{L}_t(\text{sig})$, then $\textsc{Term}(\varphi) = \mathtt{f}_{\text{finally}}(\textsc{Term}(\psi))$,
- – if $\varphi = \psi_1 \mathsf{U} \psi_2$ for $\psi_1, \psi_2 \in \mathcal{L}_t(\text{sig})$, then

  $$\textsc{Term}(\varphi) = \mathtt{f}_{\text{until}}(\textsc{Term}(\psi_1), \textsc{Term}(\psi_2))$$

  and

  – if $\varphi = \psi_1 \mathsf{P} \psi_2$ for $\psi_1, \psi_2 \in \mathcal{L}_t(\text{sig})$, then

  $$\textsc{Term}(\varphi) = \mathtt{f}_{\text{precedes}}(\textsc{Term}(\psi_1), \textsc{Term}(\psi_2)).$$

Now we construct

$$\begin{array}{rcl} \psi_1 & = & \mathtt{p}_{\text{new}}\left(\textsc{Term}\left(\chi^{(1)}_{i_0}\right), \ldots, \textsc{Term}\left(\chi^{(1)}_{i_k}\right)\right) \text{ and} \\ \psi_2 & = & \mathtt{p}_{\text{new}}\left(\textsc{Term}\left(\chi^{(2)}_{j_0}\right), \ldots, \textsc{Term}\left(\chi^{(2)}_{j_k}\right)\right) \end{array}$$

and compute

$$\psi = \text{LGS}(\psi_1, \psi_2) =: \mathtt{p}_{\text{new}}(t_1, \ldots, t_k).$$

The set of generalized literals can now be extracted from the sequence $t_1, \ldots, t_k$ of terms using the transformation $\textsc{Literal} : \mathcal{T}(\text{sig}_{\text{ext}}) \to \mathcal{L}_t(\text{sig})$ defined as follows:

- if $t = \mathtt{f}_{\mathtt{p}}(t_1, \ldots, t_n)$ for some $\mathtt{p} \in P$ and $t_1, \ldots, t_n \in \mathcal{T}(\text{sig})$, then $\textsc{Literal}(t) = \mathtt{p}(t_1, \ldots, t_n)$,
- if $t = \mathtt{f}_{\text{not}}(\bar{t})$ for some $\bar{t} \in \mathcal{T}(\text{sig}_{\text{ext}})$, then $\textsc{Literal}(t) = \mathtt{not}(\textsc{Literal}(\bar{t}))$,
- – if $t = \mathtt{f}_{\text{next}}(\bar{t})$ for some $\bar{t} \in \mathcal{T}(\text{sig}_{\text{ext}})$, then $\textsc{Literal}(t) = \mathsf{X}\textsc{Literal}(\bar{t})$,
  – if $t = \mathtt{f}_{\text{always}}(\bar{t})$ for some $\bar{t} \in \mathcal{T}(\text{sig}_{\text{ext}})$, then $\textsc{Literal}(t) = \mathsf{G}\textsc{Literal}(\bar{t})$ and

– if $t = \mathtt{f}_{\text{finally}}(\bar{t})$ for some $\bar{t} \in \mathcal{T}(\text{sig}_{\text{ext}})$, then $\textsc{Literal}(t) = \mathsf{F}\textsc{Literal}(\bar{t})$,

- – if $t = \mathtt{f}_{\text{until}}(t_1, t_2)$ for $t_1, t_2 \in \mathcal{T}(\text{sig}_{\text{ext}})$, then

$$\textsc{Literal}(t) = \textsc{Literal}(t_1)\mathsf{U}\textsc{Literal}(t_2)$$

and

– if $t = \mathtt{f}_{\text{precedes}}(t_1, t_2)$ for $t_1, t_2 \in \mathcal{T}(\text{sig}_{\text{ext}})$, then

$$\textsc{Literal}(t) = \textsc{Literal}(t_1)\mathsf{P}\textsc{Literal}(t_2).$$

It is immediately clear from the definition of the operations Term and Literal that we have for every literal φ and every term t:

$$\begin{aligned} \textsc{Literal}(\textsc{Term}(\varphi)) &= \varphi \text{ and} \\ \textsc{Term}(\textsc{Literal}(t)) &= t \end{aligned}$$

Having constructed $\psi = \mathtt{p}_{\text{new}}(t_1, \ldots, t_k)$ as described above, we extract the generalized literals by computing the following set:

$$S_C = \{\textsc{Literal}(t_1), \ldots, \textsc{Literal}(t_k)\} = \{\chi_1, \ldots, \chi_k\}.$$

By assumption we have that there is at least one positive literal χ_i. So the set of possible rules which can be extracted from S_C is nonempty. We will again use the operation Tail defined on page 126. The algorithm to construct the set or rules from S_C is described in Algorithm 6.

Theorem 6.3.2

Let C_1 and C_2 be Prolog(+T)–rules and let C be any Prolog(+T)–rule from the set computed by Algorithm 6 given inputs C_1 and C_2. Then C is a least generalization

Algorithm 6 Least Generalization of Prolog(+T)–rules

```
Input: Prolog(+T)–rules $C_1$, $C_2$
Output: set of least generalizations of $C_1$ and $C_2$
Require: $C_1 = \varphi_1 :\!- \psi_1^{(1)}, \ldots, \psi_{n_1}^{(1)}$, $C_2 = \varphi_1 :\!- \psi_1^{(2)}, \ldots, \psi_{n_2}^{(2)}$
 1: $L \leftarrow \emptyset$
 2: $S_{C_1} \leftarrow \left\{\neg\psi_1^{(1)}, \ldots, \neg\psi_{n_1}^{(1)}, \varphi_1\right\}$
 3: $S_{C_2} \leftarrow \left\{\neg\psi_1^{(2)}, \ldots, \neg\psi_{n_2}^{(2)}, \varphi_2\right\}$
 4: $S \leftarrow S_{C_1} \cup S_{S_2}$
Require: $S = \{l_1, \ldots, l_o\}$
 5: $\textsc{Co} \leftarrow \emptyset$
 6: for $i = 1, \ldots, o$ do
 7:   for $j = i + 1, \ldots, o$ do
 8:     if $i \neq j$ then
 9:       if $\textsc{Temp}(l_i) = \textsc{Temp}(l_j)$ then
10:         $\textsc{Co} \leftarrow \textsc{Co} \cup \{(l_i, l_j)\}$
11:       end if
12:     end if
13:   end for
14: end for
Require: $\textsc{Co} = \{(l_{i_1}, l_{j_1}), \ldots, (l_{i_k}, l_{j_k})\}$
15: $S_C \leftarrow \emptyset$
Require: $\mathsf{p}_{\text{new}}$ is some new predicate symbol with arity $k$
16: $\psi_1 \leftarrow \mathsf{p}_{\text{new}}\left(\textsc{Term}\left(l_{i_1}\right), \ldots, \textsc{Term}\left(l_{i_k}\right)\right)$
17: $\psi_2 \leftarrow \mathsf{p}_{\text{new}}\left(\textsc{Term}\left(l_{j_1}\right), \ldots, \textsc{Term}\left(l_{j_1}\right)\right)$
18: $\psi \leftarrow \text{LGS}(\psi_1, \psi_2)$
Require: $\psi = \mathsf{p}_{\text{new}}(t_1, \ldots, t_k)$
19: $S_C \leftarrow \{\textsc{Literal}(t_1), \ldots, \textsc{Literal}(t_k)\}$
Require: $S_C = \{l_1, \ldots, l_k\}$
20: for $i = 1, \ldots, k$ do
21:   if $l_i$ is positive then
22:     $L \leftarrow L \cup \{l_i :\!- \textsc{Tail}(S_C \setminus \{l_i\})\}$
23:   end if
24: end for
25: if $L \neq \emptyset$ then
26:   return $L$
27: else
28:   return {false}
29: end if
```

under subsumption of C_1 and C_2.

Proof. Let C_1 and C_2 be given as

$$\begin{aligned} C_1 &= \varphi_1 :- \psi_1^{(1)}, \ldots, \psi_{n_1}^{(1)} \text{ and} \\ C_2 &= \varphi_2 :- \psi_1^{(2)}, \ldots, \psi_{n_2}^{(2)}. \end{aligned}$$

We first show that every clause C computed by Algorithm 6 given input C_1 and C_2 is a generalization of C_1 and C_2. Let S be the set returned by Algorithm 6. If $S = \{\texttt{false}\}$, then the claim is immediate by definition of the ordering $\succeq_s$. Now assume that $S \neq \emptyset$. If C is any element from S, then C has been constructed from a literal $\texttt{p}_C(t_1, \ldots, t_n)$ for some predicate symbol $\texttt{p}$, some $n \in \mathbb{N}$ and terms $t_1, \ldots, t_n$. This literal has been constructed from C_1 and C_2 by application of Algorithm 4. So $p_C(t_1, \ldots, t_n)$ is indeed a least generalization of the two literals constructed from C_1 and C_2. This gives the claim of the theorem. □

As in the case of computing sets of greatest specializations we have that each element in a set returned by Algorithm 6 is semantically equivalent to the remaining elements in this set. So we will again denote *any* Prolog(T)–rule computed by Algorithm 6 given inputs C_1 and C_2 as

$$\mathrm{LGS}(C_1, C_2).$$

Example 6.3.5

Consider the following three Prolog(+T)–rules:

$$\begin{aligned} C_1 &= \texttt{Gp(a)} :- \texttt{Fq(X)}., \\ C_2 &= \texttt{p(b)} :- \texttt{not(q(X))Ur(X)}, \texttt{Fq(a)}. \text{ and} \\ C_3 &= \texttt{p(a)} :- \texttt{not(q(a))Ur(c)}. \end{aligned}$$

We have $\mathrm{LGS}(C_1, C_2) = \mathrm{LGS}(C_1, C_3) = \{\texttt{false}\}$. The computation of $\mathrm{LGS}(C_2, C_3)$ is carried out as follows: We have $\text{Co} = \{(\mathtt{p(a)}, \mathtt{p(a)}), (\mathtt{q(X)Ur(X)}, \mathtt{q(a)Ur(c)})\}$. So $\mathtt{p}_{\text{new}}$ with arity 2 is added to the signature. Then the following two literals are created:

$$\begin{aligned}\psi_1 &= \mathtt{p}_{\text{new}}\left(\mathtt{f_p(a)}, \mathtt{f}_{\text{until}}(\mathtt{f_q(X)}, \mathtt{f_r(X)})\right) \text{ and}\\ \psi_2 &= \mathtt{p}_{\text{new}}\left(\mathtt{f_p(a)}, \mathtt{f}_{\text{until}}(\mathtt{f_q(a)}, \mathtt{f_r(c)})\right).\end{aligned}$$

The least generalization of these literals is $\mathtt{p}_{\text{new}}(\mathtt{f_p(a)}, \mathtt{f}_{\text{until}}(\mathtt{f_q}(\mathtt{Z}_1), \mathtt{f_r}(\mathtt{Z}_2)))$. This gives $S_C = \{\mathtt{p(a)}, \mathtt{q}(\mathtt{Z}_1)\mathtt{Ur}(\mathtt{Z}_2)\}$ and therefore the set C of least generalizations is

$$C = \{\mathtt{p(a)} \mathrel{:-} \mathtt{not}(\mathtt{q}(\mathtt{Z}_1)\mathtt{Ur}(\mathtt{Z}_2)).,\ \mathtt{q}(\mathtt{Z}_1)\mathtt{Ur}(\mathtt{Z}_2) \mathrel{:-} \mathtt{not}(\mathtt{p(a)}).\}\,.$$

The results from this section and the section before are summarized in the following theorem.

Theorem 6.3.3

Let $\mathcal{R}$ be the set of all PROLOG(+T)–rules. Then $(\mathcal{R} \cup \{\{\mathtt{false}\}\}, \succeq_s)$ is a lattice.

This theorem enables systems to refine programs in order to fit their specifications without taking hazards of *overgeneralization* or *overspecialization*. It will turn out important in the following chapter where we will define refinement operators for PROLOG(+T)–rules.

7. Refinement Operators for Prolog(+T)–programs

Contents

7.1. Refinement Operators for PROLOG(+T)–Literals 142

7.2. Refinement Operators for Rules 147

7.2.1. Downward Refinement 150

7.2.2. Upward Refinement 152

7.3. Refinement Strategies 159

7.3.1. Elimination of Variants 162

7.3.2. Restriction to reduced Rules 163

7.3.3. Elimination of Tautologies 165

7.3.4. Premises vs. Conclusions 166

We will now consider the problem of *refining* literals and rules. In contrast to the computation of least generalizations and greatest specializations as described in the last chapter, only one object is involved now. For objects from first order logic several refinement operators have been described and studied (see [98] and [126]). We will see that the refinement operators for literals can be directly used in order to refine PROLOG(+T)–literals while the refinement operators for first order clauses have to be extended in order to be useful for the computation of refinements of PROLOG(+T)–rules. But this extension causes the number of refinements to grow very fast. So it is necessary to control the refinement process in order to avoid computational intractability. How to achieve this will be the topic of chapter 7.3.

The use of refinement operators in order to construct more special resp. more general objects from given ones dates back to 1981 when Shapiro introduced the model inference framework (see [145] and [146]). The approach posed there has gained a great interest in refinement operators although it has been shown by van der Laag (see [159]) that Shapiro's operator is (in contrast to Shapiro's arguments which shows that his proof is incorrect) not complete. However, in [159] and [98] it has been shown that complete refinement operators indeed exist.[1] [127]

Further research on refinement operators has pointed out several conditions for the existence of complete refinement operators (see [162], [161], [160], [127] and [163]). Additionally refinement operators for *theories* have been introduced (see e.g. [15]). Refinement operators for theories work on sets of clauses rather than on single clauses. This approach may yield smaller hypothesis–programs since the application of such refinement operators can be combined with techniques such as clause–deletion.

7.1. Refinement Operators for Prolog(+T)–Literals

We will now briefly resume some refinement operators for literals which have been described for first order logic atoms in [126]. The extension of these operators to PROLOG(+T)–literals is obvious and their properties carry over to PROLOG(+T). Therefore assume that the signature under consideration is $\text{sig} = (\mathcal{X}, F, P, \alpha)$ with $F = \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$.

Downward Refinement Let $\varphi \in \mathcal{L}_t(\text{sig})$ be given with $\textsc{Var}(\varphi) = \{\mathtt{Z}_1, \ldots, \mathtt{Z}_k\}$ and let $\mathtt{X}_1^{(1)}, \ldots, \mathtt{X}_{\alpha(\mathtt{f}_1)}^{(1)}, \ldots, \mathtt{X}_1^{(n)}, \ldots, \mathtt{X}_{\alpha(\mathtt{f}_n)}^{(n)}$ be a sequence of pairwise distinct variables such that $\mathtt{X}_i^{(j)} \notin \textsc{Var}(\varphi)$ for all i, j. The *downward refinement operator* $\Theta_d^{\mathcal{L}} : \mathcal{L}_t(\text{sig}) \rightarrow$

[1]The problem with Shapiro's incomplete refinement operator is simply due to the fact that it requires clauses to be reduced (see [133] and [78]). Relaxing this requirement yields on the one hand a larger search space but on the other hand it yields a complete operator.

1.	$\mathsf{GXFp}\left(\mathtt{f}\left(\mathtt{X}_1^{(1)}\right)\right)\mathsf{Up}\left(\mathtt{g}\left(\mathtt{f}\left(\mathtt{f}\left(\mathtt{X}_1^{(1)}\right)\right),\mathtt{Y}\right)\right)$
2.	$\mathsf{GXFp}\left(\mathtt{g}\left(\mathtt{X}_1^{(2)},\mathtt{X}_2^{(2)}\right)\right)\mathsf{Up}\left(\mathtt{g}\left(\mathtt{f}\left(\mathtt{g}\left(\mathtt{X}_1^{(2)},\mathtt{X}_2^{(2)}\right)\right),\mathtt{Y}\right)\right)$
3.	$\mathsf{GXFp}(\mathtt{X})\,\mathsf{Up}\left(\mathtt{g}\left(\mathtt{f}(\mathtt{X}),\mathtt{f}\left(\mathtt{X}_1^{(1)}\right)\right)\right)$
4.	$\mathsf{GXFp}(\mathtt{X})\,\mathsf{Up}\left(\mathtt{g}\left(\mathtt{f}(\mathtt{X}),\mathtt{g}\left(\mathtt{X}_1^{(2)},\mathtt{X}_2^{(2)}\right)\right)\right)$
5.	$\mathsf{GXFp}(\mathtt{a})\mathsf{Up}(\mathtt{g}(\mathtt{f}(\mathtt{a}),\mathtt{Y})), \mathsf{GXFp}(\mathtt{X})\mathsf{Up}(\mathtt{g}(\mathtt{f}(\mathtt{X}),\mathtt{a}))$
6.	$\mathsf{GXFp}(\mathtt{Y})\mathsf{Up}(\mathtt{g}(\mathtt{f}(\mathtt{Y}),\mathtt{Y})), \mathsf{GXFp}(\mathtt{X})\mathsf{Up}(\mathtt{g}(\mathtt{f}(\mathtt{X}),\mathtt{X}))$

Table 7.2.: Set of downward refinements of $\varphi = \mathsf{GXFp}(\mathtt{X})\mathsf{Up}(\mathtt{g}(\mathtt{f}(\mathtt{X}),\mathtt{Y}))$

$2^{\mathcal{L}_t(\mathrm{sig})}$ is now defined as follows:

$$\Theta_d^{\mathcal{L}}(\varphi) = \left\{\varphi\left\{\frac{\mathtt{Z}_i}{\mathtt{f}_j\left(\mathtt{X}_1^{(j)},\ldots,\mathtt{X}_{\alpha(\mathtt{f}_j)}^{(j)}\right)}\right\} \mid i=1,\ldots,k, j=1,\ldots,n\right\} \tag{7.1}$$

$$\cup \left\{\varphi\left\{\frac{\mathtt{Z}_j}{\mathtt{Z}_i}\right\} \mid i=1,\ldots,k, j=1,\ldots,k, i\neq j\right\}. \tag{7.2}$$

The set from (7.1) creates the literals which emerge from the original literal by replacing *one* variable by all possible instantiations of function symbols with (new) variables (note that replacement of variables with constant symbols is just a special case of this case) while the set from (7.2) replaces variables with other variables occurring in the original expression. The procedure carried out by the application of $\Theta_d^{\mathcal{L}}$ is summarized in Algorithm 7.

Example 7.1.1

Assume that $F = \{\mathtt{f},\mathtt{g},\mathtt{a}\}$ and $P = \{\mathtt{p}\}$ with $\alpha(\mathtt{p}) = \alpha(\mathtt{g}) = 2$, $\alpha(\mathtt{f}) = 1$ and $\alpha(\mathtt{a}) = 0$. If $\varphi = \mathsf{GXFp}(\mathtt{X})\mathsf{Up}(\mathtt{g}(\mathtt{f}(\mathtt{X}),\mathtt{Y}))$, then $\mathrm{VAR}(\varphi) = \{\mathtt{X},\mathtt{Y}\}$. The new variables to be introduced are $\mathtt{X}_1^{(1)}$,$\mathtt{X}_1^{(2)}$ and $\mathtt{X}_2^{(2)}$. The result of $\Theta_d^{\mathcal{L}}$ given input φ is summarized in Table 7.2.

It is easy to see that

1. $\Theta_d^{\mathcal{L}}$ is ideal and

2. $\left|\Theta_d^{\mathcal{L}}(\varphi)\right| \leq |\textsc{Var}(\varphi)| \cdot (|F| + |\textsc{Var}(\varphi)| - 1)$ for every $\varphi \in \mathcal{L}_t(\text{sig})$.

Algorithm 7 Downward–Refinement of PROLOG(+T)–literals

```
Input: PROLOG(+T)–literal $\varphi$ built from sig $= (\mathcal{X}, F, P, \alpha)$
Output: set of specialized literals
Require: $\textsc{Var}(\varphi) = \{\mathtt{X}_1, \dots, \mathtt{X}_k\}$
Require: $F = \{\mathtt{f}_1, \dots, \mathtt{f}_n\}$
Require: $\mathtt{X}_1^{(1)}, \dots, \mathtt{X}_{\alpha(\mathtt{f}_1)}^{(1)}, \dots, \mathtt{X}_1^{(n)}, \dots, \mathtt{X}_{\alpha(\mathtt{f}_n)}^{(n)}$ is a sequence of pairwise distinct variables
    from $\mathcal{X} \setminus \textsc{Var}(\varphi)$
 1: $Ref \leftarrow \emptyset$
 2: for $i = 1, \dots, k$ do
 3:   for $j = 1, \dots, n$ do
 4:     $\sigma \leftarrow \left\{\frac{\mathtt{X}_i}{\mathtt{f}_j\left(\mathtt{X}_1^{(j)}, \dots, \mathtt{X}_{\alpha(\mathtt{f}_j)}^{(j)}\right)}\right\}$
 5:     $Ref \leftarrow Ref \cup \{\sigma(\varphi)\}$
 6:   end for
 7: end for
 8: for $i = 1, \dots, k$ do
 9:   for $j = 1, \dots, k$ do
10:     if $i \neq j$ then
11:       $\sigma \leftarrow \left\{\frac{\mathtt{X}_i}{\mathtt{X}_j}\right\}$
12:       $Ref \leftarrow Ref \cup \{\sigma(\varphi)\}$
13:     end if
14:   end for
15: end for
16: return $Ref$
```

Upward Refinement The dual case of downward refinement is upward refinement. The upward refinement operator $\Theta_u^{\mathcal{L}}$ for PROLOG(+T)–literals which we will present now is (as the operator $\Theta_d^{\mathcal{L}}$ is) an extension of an ideal refinement operator for first order logic literals. In order to present the refinement operator we need some more formal concepts.

First we will define the mapping $\textsc{Terms} : \mathcal{L}_t(\text{sig}) \cup \mathcal{T}(\text{sig}) \to 2^{\mathcal{T}(\text{sig})}$ which returns all terms which occur in a term respectively in a literal:

1. if $t = \mathtt{X} \in \mathcal{X}$, then $\textsc{Terms}(t) = \{\mathtt{X}\}$,

2. if $t = \mathtt{f}(t_1, \ldots, t_n)$ for some function symbol $\mathtt{f}$ with arity n and terms $t_1, \ldots, t_n$, then $\textsc{Terms}(t) = \{t\} \cup \bigcup_{i=1}^{n} \textsc{Terms}(t_i)$,
3. if $\varphi = \mathtt{p}(t_1, \ldots, t_n)$ for some predicate symbol $\mathtt{p}$ with arity n and terms $t_1, \ldots, t_n$, then $\textsc{Terms}(\varphi) = \bigcup_{i=1}^{n} \textsc{Terms}(t_i)$,
4. if $\varphi = \mathtt{not}(\psi)$ for some $\psi \in \mathcal{L}_t(\text{sig})$, then $\textsc{Terms}(\varphi) = \textsc{Terms}(\psi)$,
5. if $\varphi = \oplus\psi$ for some $\psi \in \mathcal{L}_t(\text{sig})$ and some $\oplus \in \{\mathsf{X}, \mathsf{G}, \mathsf{F}\}$, then $\textsc{Terms}(\varphi) = \textsc{Terms}(\psi)$ and
6. if $\varphi = \psi_1 \oplus \psi_2$ for $\psi_1, \psi_2 \in \mathcal{L}_t(\text{sig})$ and $\oplus \in \{\mathsf{U}, \mathsf{P}\}$, then $\textsc{Terms}(\varphi) = \textsc{Terms}(\psi_1) \cup \textsc{Terms}(\psi_2)$.

We will call a term $t \in \mathcal{T}(\text{sig})$ *simple* if $t = \mathtt{f}(\mathtt{X}_1, \ldots, \mathtt{X}_n)$ for a function symbol $\mathtt{f}$ with arity n and variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$ such that $\mathtt{X}_i \neq \mathtt{X}_j$ for $i \neq j$.

Now let o_1 and o_2 be any objects (terms or literals). The set of all *occurrences* of o_1 in o_2 is defined as

$$\textsc{Occ}(o_1, o_2) = \{p \in \textsc{Pos}(o_2) \mid o_1|_p = o_2\}\,.$$

An occurrence p_1 of an object o_1 is said to be *inside* an occurrence p_2 of another object o_2 if there is $p \in \mathbb{N}^*$ such that $p_2 p = p_1$.

Recall that for a literal φ, $p \in \textsc{Pos}(\varphi)$ and a term t the literal $\varphi[t]_p$ is defined as the literal which emerges from φ by replacing the term at position p with t. Similarly for $p_1, \ldots, p_k \in \textsc{Pos}(\varphi)$ and $k > 1$, the literal $\varphi[t]_{p_1,\ldots,p_k}$ emerges from φ by replacing the terms at positions $p_1, \ldots, p_k$ with t.

These concepts enable the definition of the upward refinement operator $\Theta_u^{\mathcal{L}}$ for FOLTL–literals.

$$
\begin{aligned}
\Theta_u^{\mathcal{L}}(\varphi) \quad &= \quad \underbrace{\left\{ \varphi\,[\mathtt{Z}]_{p_1,\ldots,p_k} \;\middle|\; \begin{array}{c} \emptyset \neq \{p_1,\ldots,p_k\} = \textsc{Occ}(t,\varphi) \text{ for every simple } t \\ \text{such that for every } \mathtt{X} \in \textsc{Var}(t) \text{ every } p \in \textsc{Occ}(\mathtt{X},t) \\ \text{is inside one of the } p_j \text{ and } \mathtt{Z} \notin \textsc{Var}(\varphi) \end{array} \right\}}_{=:S_1} \\
&\cup \quad \underbrace{\left\{ \varphi\,[\mathtt{Z}]_{p_1,\ldots,p_k} \;\middle|\; \begin{array}{c} \text{for every } \mathtt{a} \in F \text{ such that } \alpha(\mathtt{a}) = 0 \text{ and every} \\ \emptyset \neq \{p_1,\ldots,p_k\} \subseteq \textsc{Occ}(\mathtt{a},\varphi) \text{ and some } \mathtt{Z} \notin \textsc{Var}(\varphi) \end{array} \right\}}_{=:S_2} \\
&\cup \quad \underbrace{\left\{ \varphi\,[\mathtt{Z}]_{p_1,\ldots,p_k} \;\middle|\; \begin{array}{c} \text{for every } \mathtt{X} \in \textsc{Var}(\varphi), \text{ every set} \\ \emptyset \neq \{p_1,\ldots,p_k\} \subset \textsc{Occ}(\mathtt{X},\varphi) \text{ and some } \mathtt{Z} \notin \textsc{Var}(\varphi) \end{array} \right\}}_{=:S_3}
\end{aligned}
$$

The estimation of the number of elements in $\Theta_u^{\mathcal{L}}(\varphi)$ is not that easy and we are only able to present a very weak estimation. We have

$$
\begin{aligned}
|S_1| &\leq |\textsc{Terms}(\varphi)|, \\
|S_2| &\leq |F| \cdot \left(2^{|\textsc{Pos}(\varphi)|} - 1\right) \text{ and} \\
|S_3| &\leq |\textsc{Var}(\varphi)| \cdot \left(2^{|\textsc{Pos}(\varphi)|} - 2\right),
\end{aligned}
$$

which gives

$$
\begin{aligned}
\left|\Theta_u^{\mathcal{L}}(\varphi)\right| &\leq |S_1| + |S_2| + |S_3| \\
&\leq |\textsc{Terms}(\varphi)| + |F| \cdot \left(2^{|\textsc{Pos}(\varphi)|} - 1\right) + |\textsc{Var}(\varphi)| \cdot \left(2^{|\textsc{Pos}(\varphi)|} - 2\right).
\end{aligned}
$$

This estimation is not very precise as the following example shows. But it is not yet clear how a better estimation might be derived from the definition of $\Theta_u^{\mathcal{L}}$ without taking the structure of the involved terms into account.

Example 7.1.2

Consider the signature sig $= (\mathcal{X}, \{\mathtt{f}, \mathtt{a}\}, \{\mathtt{p}\}, \alpha)$ with $\alpha(\mathtt{a}) = 0$, $\alpha(\mathtt{f}) = 2$ and $\alpha(\mathtt{p}) = 3$ and the literal $\varphi = \mathsf{FGp}(\mathtt{X}_1, \mathtt{f}(\mathtt{a}, \mathtt{f}(\mathtt{X}_1, \mathtt{f}(\mathtt{a}, \mathtt{f}(\mathtt{a}, \mathtt{a})))), \mathtt{f}(\mathtt{X}_1, \mathtt{X}_2))$. Then the only simple term in $\textsc{Terms}(\varphi)$ is $\mathtt{f}(\mathtt{X}_2, \mathtt{X}_3)$ at position $p = 113$. So $\Theta_u^{\mathcal{L}}(\varphi)$ contains the literal $\mathsf{FGp}(\mathtt{X}_1, \mathtt{f}(\mathtt{a}, \mathtt{f}(\mathtt{a}, \mathtt{f}(\mathtt{a}, \mathtt{f}(\mathtt{a}, \mathtt{a})))), \mathtt{Z})$. Since $\textsc{Var}(\varphi) = \{\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3\}$ we have that $\textsc{Occ}(\mathtt{X}_1, \varphi) = \{111\}$, $\textsc{Occ}(\mathtt{X}_2, \varphi) = \{1131\}$ and $\textsc{Occ}(\mathtt{X}_3, \varphi) = \{1132\}$. None of these sets has a nonempty proper subset so $S_3 = \emptyset$ in this case. Furthermore $\textsc{Occ}(\mathtt{a}, \varphi) = \{1121, 11221, 112221, 1122221, 1122222\}$, so there are 31 possible literals which might be added by $\Theta_u^{\mathcal{L}}$. So the overall size of $\Theta_u^{\mathcal{L}}(\varphi)$ is 32 while the above estimation yields

$$\begin{aligned}
\left|\Theta_u^{\mathcal{L}}(\varphi)\right| &\leq |S_1| + |S_2| + |S_3| \\
&\leq |\textsc{Terms}(\varphi)| + |F| \cdot \left(2^{|\textsc{Pos}(\varphi)|} - 1\right) + |\textsc{Var}(\varphi)| \cdot \left(2^{|\textsc{Pos}(\varphi)|} - 2\right) \\
&= 9 + 2 \cdot \left(2^{16} - 1\right) + 3 \cdot \left(2^{16} - 2\right) \\
&= 327681
\end{aligned}$$

The complete set of refined literals is listed in Table 7.3.

Since the original refinement operator for first order logic literals is ideal, $\Theta_u^{\mathcal{L}}$ is also ideal. The procedure for computing $\Theta_u^{\mathcal{L}}(\varphi)$ is summarized in Algorithm 8.

7.2. Refinement Operators for Rules

We will now present adapted versions of *classical* refinement operators for sets of literals, i.e. for $\textsc{Prolog}(+\mathrm{T})$–rules. As in the case of refining literals, there is in general more than one refinement of an input rule. So we will have a set of rules as the result of a refinement operation. Each of these rules is a set from which we may construct one or more rules each of which is a refinement of the original rule.

1.	$FGp(X_1,f(a,f(a,f(a,f(a,a)))),Z)$
2.	$FGp(X_1,f(a,f(a,f(a,f(a,Z)))),f(X_1,X_2))$
3.	$FGp(X_1,f(a,f(a,f(a,f(Z,a)))),f(X_1,X_2))$
4.	$FGp(X_1,f(a,f(a,f(a,f(Z,Z)))),f(X_1,X_2))$
5.	$FGp(X_1,f(a,f(a,f(Z,f(a,a)))),f(X_1,X_2))$
6.	$FGp(X_1,f(a,f(a,f(Z,f(a,Z)))),f(X_1,X_2))$
7.	$FGp(X_1,f(a,f(a,f(Z,f(Z,a)))),f(X_1,X_2))$
8.	$FGp(X_1,f(a,f(a,f(Z,f(Z,Z)))),f(X_1,X_2))$
9.	$FGp(X_1,f(a,f(Z,f(a,f(a,a)))),f(X_1,X_2))$
10.	$FGp(X_1,f(a,f(Z,f(a,f(a,Z)))),f(X_1,X_2))$
11.	$FGp(X_1,f(a,f(Z,f(a,f(Z,a)))),f(X_1,X_2))$
12.	$FGp(X_1,f(a,f(Z,f(a,f(Z,Z)))),f(X_1,X_2))$
13.	$FGp(X_1,f(a,f(Z,f(Z,f(a,a)))),f(X_1,X_2))$
14.	$FGp(X_1,f(a,f(Z,f(Z,a(a,Z)))),f(X_1,X_2))$
15.	$FGp(X_1,f(a,f(Z,f(Z,f(Z,a)))),f(X_1,X_2))$
16.	$FGp(X_1,f(a,f(Z,f(Z,f(Z,Z)))),f(X_1,X_2))$
17.	$FGp(X_1,f(Z,f(a,f(a,f(a,a)))),f(X_1,X_2))$
18.	$FGp(X_1,f(Z,f(a,f(a,f(a,z)))),f(X_1,X_2))$
19.	$FGp(X_1,f(Z,f(a,f(a,f(Z,a)))),f(X_1,X_2))$
20.	$FGp(X_1,f(Z,f(a,f(a,f(Z,Z)))),f(X_1,X_2))$
21.	$FGp(X_1,f(Z,f(a,f(Z,f(a,a)))),f(X_1,X_2))$
22.	$FGp(X_1,f(Z,f(a,f(Z,f(a,Z)))),f(X_1,X_2))$
23.	$FGp(X_1,f(Z,f(a,f(a,f(Z,a)))),f(X_1,X_2))$
24.	$FGp(X_1,f(Z,f(a,f(a,f(Z,Z)))),f(X_1,X_2))$
25.	$FGp(X_1,f(Z,f(a,f(Z,f(a,a)))),f(X_1,X_2))$
26.	$FGp(X_1,f(Z,f(a,f(Z,f(a,Z)))),f(X_1,X_2))$
27.	$FGp(X_1,f(Z,f(a,f(Z,f(Z,a)))),f(X_1,X_2))$
28.	$FGp(X_1,f(Z,f(a,f(Z,f(Z,Z)))),f(X_1,X_2))$
29.	$FGp(X_1,f(Z,f(Z,f(Z,f(a,a)))),f(X_1,X_2))$
30.	$FGp(X_1,f(Z,f(Z,f(Z,f(a,z)))),f(X_1,X_2))$
31.	$FGp(X_1,f(Z,f(Z,f(Z,f(Z,a)))),f(X_1,X_2))$
32.	$FGp(X_1,f(Z,f(Z,f(Z,f(Z,Z)))),f(X_1,X_2))$

Table 7.3.: Set of upward refinements for $\varphi = FGp(X_1,f(a,f(X_1,f(a,f(a,a)))),f(X_1,X_2))$

Algorithm 8 Upward–Refinement of PROLOG(+T)–literals

```
Input: PROLOG(+T)–literal $\varphi$
Output: set of generalized literals
Require: $\textsc{Var}(C) = \{\mathtt{X}_1, \ldots, \mathtt{X}_k\}$
Require: $\mathtt{Z} \in \mathcal{X} \setminus \{\mathtt{X}_1, \ldots, \mathtt{X}_k\}$
Require: $\textsc{Terms}(\varphi) = \{t_1, \ldots, t_n\}$
 1: $\mathit{Ref} \leftarrow \emptyset$
 2: for $i = 1, \ldots, n$ do
 3:   if $t_i$ is simple then
Require:   $\textsc{Var}(t_i) = \{\mathtt{Z}_1, \ldots, \mathtt{Z}_l\}$
 4:     for $o = 1, \ldots, l$ do
 5:       if each $p \in \textsc{Occ}(\mathtt{Z}_o, t_i)$ is inside of one element from $\textsc{Occ}(t_i, \varphi)$ then
Require:       $\{p_1, \ldots, p_m\} = \textsc{Occ}(t_i, \varphi)$
 6:         $\mathit{Ref} \leftarrow \mathit{Ref} \cup \{\varphi[\mathtt{Z}]_{p_1,\ldots,p_m}\}$
 7:       end if
 8:     end for
 9:   end if
10: end for
Require: $F = \{\mathtt{f}_1, \ldots, \mathtt{f}_{|F|}$
11: for $i = 1, \ldots, |F|$ do
12:   if $\alpha(\mathtt{f}_i) = 0$ then
13:     for each $\emptyset \neq \{p_1, \ldots, p_m\} \subseteq \textsc{Occ}(f_i, \varphi)$ do
14:       $\mathit{Ref} \leftarrow \mathit{Ref} \cup \{\varphi[\mathtt{Z}]_{p_1,\ldots,p_m}\}$
15:     end for
16:   end if
17: end for
18: for $i = 1, \ldots, k$ do
19:   for each $\emptyset \neq \{p_1, \ldots, p_m\} \subset \textsc{Occ}(\mathtt{X}_i, \varphi)$ do
20:     $\mathit{Ref} \leftarrow \mathit{Ref} \cup \{\varphi[\mathtt{Z}]p_1, \ldots, p_m\}$
21:   end for
22: end for
23: return $\mathit{Ref}$
```

In the following sections we will sometimes refer to certain refinement operators which have been introduced for first order logic literals respectively clauses. A general description of refinement operators can be found in [98]. In particular we will refer to the operators $\rho_{\mathcal{A}}$ (downward refinement of atoms), $\delta_{\mathcal{A}}$ (upward refinement of atoms), ρ_l (downward refinement of clauses) and δ_u (upward refinement of clauses) from [126].

7.2.1. Downward Refinement

The basic idea of *downward refinement* of Prolog(+T)–rules is first to consider the set S_C induced by a rule C and then to add certain literals to S_C. Additionally we will replace variables with terms and variables with other variables from the original rule as in the case of the operator $\Theta_d^{\mathcal{L}}$.

The original downward refinement operator presented in [98] only treats first order clauses. So in order to construct refinements of Prolog(+T)–rules we have to deal with the temporal operators. This will be done as follows:

- whenever a literal ψ is contained in the original set S_C, the set of refinements contains $S_C \cup \{\oplus\psi\}$ and $S_C \cup \{\mathtt{not}(\oplus\psi)\}$ for $\oplus \in \{\mathsf{X}, \mathsf{G}, \mathsf{F}\}$ and
- whenever two literals ψ_1 and ψ_2 are contained in S_C, the set of refinements contains both $S_C \cup \{\psi_1 \oplus \psi_2\}$ and $S_C \cup \{\mathtt{not}(\psi_1 \oplus \psi_2)\}$ for $\oplus \in \{\mathsf{U}, \mathsf{P}\}$.

From the resulting set of literals we will extract those rules which can be written using a head literal which is positive.

The operator $\Theta_d^{\mathcal{R}} : 2^{\mathcal{L}_t(\mathrm{sig})} \rightarrow 2^{2^{\mathcal{L}_t(\mathrm{sig})}}$ is therefore defined as follows:

$$\Theta_d^{\mathcal{R}}(S_C) \quad = \quad \left\{ S_C \left\{ \frac{\mathtt{Z}}{\mathtt{f}(\mathtt{X}_1, \ldots, \mathtt{X}_l)} \right\} \;\middle|\; \begin{array}{c} \mathtt{Z} \in \mathrm{Var}(C), \mathtt{f} \in F, \alpha(\mathtt{f}) = l, \\ \mathtt{X}_1, \ldots, \mathtt{X}_l \notin \mathrm{Var}(C), \\ \mathtt{X}_i \neq \mathtt{X}_j \text{ for } i \neq j \end{array} \right\} \tag{7.3}$$

$$\cup \quad \left\{S_C\left\{\frac{\mathtt{Z}}{\mathtt{X}}\right\} \mid \mathtt{X},\mathtt{Z} \in \textsc{Var}(C), \mathtt{X} \neq \mathtt{Z}\right\} \tag{7.4}$$

$$\cup \quad \left\{S_C \cup \{\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_l)\} \mid \begin{array}{r} \mathtt{p} \in P, \alpha(\mathtt{p}) = l, \mathtt{X}_1,\ldots,\mathtt{X}_l \notin \textsc{Var}(C) \\ \mathtt{X}_i \neq \mathtt{X}_j \text{ for } i \neq j \end{array}\right\} \tag{7.5}$$

$$\cup \quad \left\{S_C \cup \{\mathtt{not}(\mathtt{p}(\mathtt{X}_1,\ldots,\mathtt{X}_l))\} \mid \begin{array}{r} \mathtt{p} \in P, \alpha(\mathtt{p}) = l, \\ \mathtt{X}_1,\ldots,\mathtt{X}_l \notin \textsc{Var}(C) \\ \mathtt{X}_i \neq \mathtt{X}_j \text{ for } i \neq j \end{array}\right\} \tag{7.6}$$

$$\cup \quad \{S_C \cup \{\mathsf{X}\psi\} \mid \psi \in S_C\} \cup \{S_C \cup \{\mathtt{not}(\mathsf{X}\psi)\} \mid \psi \in S_C\} \tag{7.7}$$

$$\cup \quad \{S_C \cup \{\mathsf{G}\psi\} \mid \psi \in S_C\} \cup \{S_C \cup \{\mathtt{not}(\mathsf{G}\psi)\} \mid \psi \in S_C\} \tag{7.8}$$

$$\cup \quad \{S_C \cup \{\mathsf{F}\psi\} \mid \psi \in S_C\} \cup \{S_C \cup \{\mathtt{not}(\mathsf{F}\psi)\} \mid \psi \in S_C\} \tag{7.9}$$

$$\cup \quad \{S_C \cup \{\psi_1 \mathsf{U} \psi_2\} \mid \psi_1, \psi_2 \in S_C\} \tag{7.10}$$

$$\cup \quad \{S_C \cup \{\mathtt{not}(\psi_1 \mathsf{U} \psi_2)\} \mid \psi_1, \psi_2 \in S_C\} \tag{7.11}$$

$$\cup \quad \{S_C \cup \{\psi_1 \mathsf{P} \psi_2\} \mid \psi_1, \psi_2 \in S_C\} \tag{7.12}$$

$$\cup \quad \{S_C \cup \{\mathtt{not}(\psi_1 \mathsf{P} \psi_2)\} \mid \psi_1, \psi_2 \in S_C\}\,. \tag{7.13}$$

The line (7.3) generates all sets of literals which emerge from the original set by instantiating variables with terms. This construction is an obvious extension of the construction from (7.1). Similarly the second line (7.4) adds sets of literals in which single variables have been replaced by other variables from the original set. The lines (7.5) and (7.6) add new literals to the original set of literals. Finally in lines (7.7) to (7.12) temporal literals built up from literals of the original set are added as described above. Obviously the resulting set is subsumed by the original set. Furthermore the literals which have been added are general enough to be instantiated to more special literals. The complete procedure is summarized in Algorithm 9.

We illustrate the upward refinement of Prolog(+T)–rules in the following example.

Example 7.2.1

Let sig $= (\mathcal{X}, F, P, \alpha)$ with $F = \{\mathtt{f},\mathtt{g},\mathtt{a}\}$, $P = \{\mathtt{p},\mathtt{q}\}$ and $\alpha(\mathtt{f}) = 1$, $\alpha(\mathtt{g}) = 2$, $\alpha(\mathtt{a}) = 0$,

$\alpha(\mathtt{p}) = 1$ and $\alpha(\mathtt{q}) = 2$ be given. If $C = \mathtt{Gp(X_1)}$:–$\mathtt{Fq(X_1, f(X_2)), q(a, a)Up(a)}$., then the set of refined rules is as listed in Tables 7.5, 7.7 and 7.9.

Theorem 7.2.1

$\Theta_d^{\mathcal{R}}$ is locally finite and complete.

Proof. Locally finiteness is immediately by definition of $\Theta_d^{\mathcal{R}}$. For the completeness the proof relies on the completeness of the restriction of $\Theta_d^{\mathcal{R}}$ to first order logic clauses. This has been shown in [126]. Since every possible *most general* (see section 7.2.2 for a formal definition of most general literals) temporal literal is added to the set of refinements (lines 17–22) the completeness carries over to PROLOG(+T)–rules. □

7.2.2. Upward Refinement

As having done for PROLOG(+T)–literals we will now describe how to refine PROLOG(+T)–rules *upward*. We will see that the definition of an upward refinement operator for first order logic clauses does not need to be changed. This is due to the fact that upward refinement of rules is in some sense *easier* than downward refinement since an upward refinement operator does not need to capture all possible cases of temporal literals which might be added. Therefore the upward refinement operator $\Theta_u^{\mathcal{R}}$ to be introduced is more tractable than the operator $\Theta_d^{\mathcal{R}}$. But before we need some more definitions.

Definition 7.2.1

Let sig $= (\mathcal{X}, F, P, \alpha)$ be a signature, let $\varphi \in \mathcal{L}_t(\text{sig})$ be a literal and let C be a PROLOG(+T)–rule over sig. Then φ is called *most general* with respect to C if $\textsc{Terms}(\varphi) = \{\mathtt{X}_1, \ldots, \mathtt{X}_n\} \subseteq \mathcal{X}$ and

1. $\varphi = \mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_n)$ for some $\mathtt{p} \in P$ with $\alpha(\mathtt{p}) = n$ and $\{\mathtt{X}_1, \ldots, \mathtt{X}_n\} \cap \textsc{Var}(C) = \emptyset$ or

1.	Gp(a) :–Fq(X_1, f(X_2)), q(a, a)Up(a).
2.	Gp(X_1) :–Fq(X_1, f(a)), q(a, a)Up(a).
3.	Gp(f($\bar{X}_1$)) :–Fq(f($\bar{X}_1$), f(X_2)), q(a, a)Up(q).
4.	Gp(X_1) :–Fq(X_1, f(f($\bar{X}_1$))), q(a, a)Up(a).
5.	Gp(q($\bar{X}_1$, $\bar{X}_2$)) :–Fq(g($\bar{X}_1$, $\bar{X}_2$), f(X_2)), q(a, a)Up(a).
6.	Gp(X_1) :–Fq(X_1, f(g($\bar{X}_1$, $\bar{X}_2$))), q(a, a)Up(a).
7.	Gp(X_1) :–Fq(X_1, f(X_1)), q(a, a)Up(a).
8.	Gp(X_2) :–Fq(X_2, f(X_2)), q(a, a)Up(a).
9.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(p($\bar{X}_1$)).
10.	p(X_1) :–not(Gp(X_1)), Fq(X_1, f(X_2)), g(a, a)Up(a).
11.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(q($\bar{X}_1$, $\bar{X}_2$)).
12.	q($\bar{X}_1$, $\bar{X}_2$) :–not(Gp(X_1)), Fq(X_1, f(X_2)), q(a, a)Up(a).
13.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), p($\bar{X}_1$).
14.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), q($\bar{X}_1$, $\bar{X}_2$).
15.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Xnot(Fq(X_1, f(X_2)))).
16.	Xnot(Fq(X_1, f(X_2))) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
17.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Xnot(q(a, a)Up(a))).
18.	Xnot(q(a, a)Up(a)) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
19.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(XGp(X_1)).
20.	XGp(X_1)) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
21.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), Xnot(Fq(X_1, f(X_2))).
22.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), Xnot(q(a, a)Up(a)).
23.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), XGp(X_1).
24.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gnot(Fq(X_1, f(X_2)))).
25.	Gnot(Fq(X_1, f(X_2))) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
26.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gnot(q(a, a)Up(a))).
27.	Gnot(q(a, a)Up(a)) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
28.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(GGp(X_1)).
29.	GGp(X_1)) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
30.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), Gnot(Fq(X_1, f(X_2))).
31.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), Gnot(q(a, a)Up(a)).
32.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), GGp(X_1).
33.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Fnot(Fq(X_1, f(X_2)))).
34.	Fnot(Fq(X_1, f(X_2))) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
35.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Fnot(q(a, a)Up(a))).
36.	Fnot(q(a, a)Up(a)) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
37.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(FGp(X_1)).
38.	FGp(X_1)) :–Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
39.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), Fnot(Fq(X_1, f(X_2))).
40.	Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a), Fnot(q(a, a)Up(a)).

Table 7.5.: Set of downward refinements of C = Gp(X_1) :–Fq(X_1, f(X_2)), q(a, a)Up(a). (part 1)

41.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), FGp(X_1).
42.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)UGp(X_1)).
43.	Gp(X_1)UGp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
44.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)Unot(Fq(X_1, f(X_2)))).
45.	Gp(X_1)Unot(Fq(X_1, f(X_2))) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
46.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)Unot(q(a, a)Up(a))).
47.	Gp(X_1)Unot(q(a, a)Up(a)) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
48.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(Fq(X_1, f(X_2)))UGp(X_1)).
49.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(Fq(X_1, f(X_2)))Unot(Fq(X_1, f(X_2)))).
50.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(Fq(X_1, f(X_2)))Unot(q(a, a)Up(a))).
51.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(q(a, a)Up(a)UGp(X_1)).
52.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(q(a, a)Up(a))Unot(Fq(X_1, f(X_2)))).
53.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(q(a, a)Up(a))Unot(q(a, a)Up(a))).
54.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), Gp(X_1)UGp(X_1).
55.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), Gp(X_1)Unot(Fq(X_1, f(X_2))).
56.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), Gp(X_1)Unot(q(a, a)Up(a)).
57.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Fq(X_1, f(X_2)))UGp(X_1).
58.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Fq(X_1, f(X_2)))Unot(Fq(X_1, f(X_2))).
59.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Fq(X_1, f(X_2)))Unot(q(a, a)Up(a)).
60.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(q(a, a)Up(a)UGp(X_1).
61.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(q(a, a)Up(a))Unot(Fq(X_1, f(X_2))).
62.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(q(a, a)Up(a))Unot(q(a, a)Up(a)).
63.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)PGp(X_1)).
64.	Gp(X_1)PGp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
65.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)Pnot(Fq(X_1, f(X_2)))).
66.	Gp(X_1)Pnot(Fq(X_1, f(X_2))) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
67.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)Pnot(q(a, a)Up(a))).
68.	Gp(X_1)Pnot(q(a, a)Up(a)) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Gp(X_1)).
69.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(Fq(X_1, f(X_2)))PGp(X_1)).
70.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(Fq(X_1, f(X_2)))Pnot(Fq(X_1, f(X_2)))).
71.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(Fq(X_1, f(X_2)))Pnot(q(a, a)Up(a))).
72.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(q(a, a)Up(a)PGp(X_1)).
73.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(q(a, a)Up(a))Pnot(Fq(X_1, f(X_2)))).
74.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(not(q(a, a)Up(a))Pnot(q(a, a)Up(a))).
75.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), Gp(X_1)PGp(X_1).
76.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), Gp(X_1)Pnot(Fq(X_1, f(X_2))).
77.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), Gp(X_1)Pnot(q(a, a)Up(a)).
78.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Fq(X_1, f(X_2)))PGp(X_1).
79.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Fq(X_1, f(X_2)))Pnot(Fq(X_1, f(X_2))).
80.	Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a), not(Fq(X_1, f(X_2)))Pnot(q(a, a)Up(a)).

Table 7.7.: Set of downward refinements of C = Gp(X_1) :−Fq(X_1, f(X_2)), q(a, a)Up(a). (part 1)

Algorithm 9 Downward–Refinement of PROLOG(+T)–rules

```
Input: PROLOG(+T)–rule $C = \varphi :\!- \psi_1, \ldots, \psi_n$.
Output: set of specialized rules
Require: $C = \varphi :\!- \psi_1, \ldots, \psi_n$.
 1: $S_C \leftarrow \{\varphi, \neg\psi_1, \ldots, \neg\psi_n\}$
 2: $Ref \leftarrow \emptyset$ {set of sets of literals}
Require: $S_C = \{\chi_0, \ldots, \chi_n\}$, $\textsc{Var}(C) = \{\mathtt{X}_1, \ldots, \mathtt{X}_k\}$, sig $= (\mathcal{X}, F, P, \alpha)$
Require: $F = \left\{f_1, \ldots, f_{|F|}\right\}$
Require: $\overline{\mathtt{X}}_1, \overline{\mathtt{X}}_2, \ldots, \overline{\mathtt{X}}_i, \ldots$ new variables, pairwise distinct
 3: for $i = 1, \ldots, k$ do
 4:   for $j = 1, \ldots, |F|$ do
 5:     $Ref \leftarrow Ref \cup \left\{ S_C \left\{ \frac{\mathtt{X}_i}{\mathtt{f}_j\left(\overline{\mathtt{X}}_1, \ldots, \overline{\mathtt{X}}_{\alpha(\mathtt{f}_j)}\right)} \right\} \right\}$
 6:   end for
 7: end for
 8: for $i = 1, \ldots, k$ do
 9:   for $j = 1, \ldots, k$ do
10:     if $i \neq j$ then
11:       $Ref \leftarrow Ref \cup \left\{ S_C \left\{ \frac{\mathtt{X}_i}{\mathtt{X}_j} \right\} \right\}$
12:     end if
13:   end for
14: end for
Require: $P = \{\mathtt{p}_1, \ldots, \mathtt{p}_m\}$
15: for $i = 1, \ldots, m$ do
16:   $Ref \leftarrow Ref \cup \left\{ S_C \cup \left\{ \mathtt{p}_i\left(\overline{\mathtt{X}}_1, \ldots, \overline{\mathtt{X}}_{\alpha(\mathtt{p}_i)}\right) \right\} \right\} \cup \left\{ S_C \cup \left\{ \mathtt{not}\left(\mathtt{p}_i\left(\overline{\mathtt{X}}_1, \ldots, \overline{\mathtt{X}}_{\alpha(\mathtt{p}_i)}\right)\right) \right\} \right\}$
17:   $Ref \leftarrow Ref \cup \{\mathsf{X}\psi \mid \psi \in S_C\} \cup \{\mathtt{not(}\mathsf{X}\psi\mathtt{)} \mid \psi \in S_C\}$
18:   $Ref \leftarrow Ref \cup \{\mathsf{G}\psi \mid \psi \in S_C\} \cup \{\mathtt{not(}\mathsf{G}\psi\mathtt{)} \mid \psi \in S_C\}$
19:   $Ref \leftarrow Ref \cup \{\mathsf{F}\psi \mid \psi \in S_C\} \cup \{\mathtt{not(}\mathsf{F}\psi\mathtt{)} \mid \psi \in S_C\}$
20:   $Ref \leftarrow Ref \cup \{\psi_1 \mathsf{U} \psi_2 \mid \psi_1, \psi_2 \in S_C\} \cup \{\mathtt{not(}\psi_1 \mathsf{U} \psi_2\mathtt{)} \mid \psi_1, \psi_2 \in S_C\}$
21:   $Ref \leftarrow Ref \cup \{\psi_1 \mathsf{P} \psi_2 \mid \psi_1, \psi_2 \in S_C\} \cup \{\mathtt{not(}\psi_1 \mathsf{P} \psi_2\mathtt{)} \mid \psi_1, \psi_2 \in S_C\}$
22: end for
Require: $Ref = \{S_1, \ldots, S_o\}$
23: $R \leftarrow \emptyset$ {set of rules}
24: for $i = 1, \ldots, o$ do
Require: $S_i = \left\{ l_1^{(i)}, \ldots, l_{n_i}^{(i)} \right\}$
25:   for $j = 1, \ldots, n_i$ do
26:     if $l_j$ is positive then
27:       $R \leftarrow R \cup \{l_j :\!- \textsc{Tail}(S_i \setminus \{l_j\}).\}$
28:     end if
29:   end for
30: end for
31: return $R$
```

81.	$\mathtt{Gp(X_1)}$:– $\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a), not(q(a,a)Up(a)PGp(X_1)}$.
82.	$\mathtt{Gp(X_1)}$:– $\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a), not(q(a,a)Up(a))Pnot(Fq(X_1, f(X_2)))}$.
83.	$\mathtt{Gp(X_1)}$:– $\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a), not(q(a,a)Up(a))Pnot(q(a,a)Up(a))}$.

Table 7.9.: Set of downward refinements of $C = \mathtt{Gp(X_1)}$:– $\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a)}$. (part 3)

2. $\varphi = \mathtt{not(p(X_1, \ldots, X_n))}$ for some $\mathtt{p} \in P$ with $\alpha(\mathtt{p}) = n$ and $\{\mathtt{X_1}, \ldots, \mathtt{X}_n\} \cap \textsc{Var}(C) = \emptyset$ or
3. $\varphi = \oplus\psi$ for $\oplus \in \{\mathsf{X}, \mathsf{G}, \mathsf{F}\}$ and some most general literal ψ or
4. $\varphi = \psi_1 \oplus \psi_2$ for $\oplus \in \{\mathsf{U}, \mathsf{P}\}$ and most general literals ψ_1, ψ_2 such that $\textsc{Var}(\psi_1) \cap \textsc{Var}(\psi_2) = \emptyset$.

It will be necessary to consider rules which may contain literals *more than once*. In this case, treating rules as sets of literals is not adequate. Therefore we will introduce sequences of literals which are computed by *duplicating* certain literals from the underlying set.

Definition 7.2.2 (Duplication of Literals, e.g. [126])
Let $S_C = \{\varphi_1, \ldots, \varphi_n\}$ be a set of PROLOG(+T)–literals and let $t \in \textsc{Terms}(S_C)$ be a term. Then the sequence $dup(S_C, t)$ is defined as

$$dup(S_C, t) = (\underbrace{\varphi_1, \ldots, \varphi_1}_{2^{|\textsc{Occ}(t,\varphi_1)|} \text{ times}}, \ldots, \underbrace{\varphi_n, \ldots, \varphi_n}_{2^{|\textsc{Occ}(t,\varphi_n)|} \text{ times}}).$$

Positions of such tuples of literals are defined as expected:

$$\textsc{Pos}(\varphi_1, \ldots, \varphi_n) = \bigcup_{i=1}^{n} \{ip \mid p \in \textsc{Pos}(\varphi_i)\}.$$

Similarly for $p_1, \ldots, p_k \in \textsc{Pos}(\varphi_1, \ldots, \varphi_n)$ and $t \in \mathcal{T}(\text{sig})$ we define

$$(\varphi_1, \ldots, \varphi_n)[t]_{p_1,\ldots,p_k} = \left(\varphi_1[t]_{p^{(1)}_{i^{(1)}_0},\ldots,p^{(1)}_{i^{(1)}_{m_1}}}, \ldots, \varphi_n[t]_{p^{(n)}_{i^{(n)}_0},\ldots,p^{(n)}_{i^{(n)}_{m_n}}} \right),$$

where for each j the set $\left\{ jp^{(j)}_{i^{(j)}_0}, \ldots, jp^{(j)}_{i^{(j)}_{m_j}} \right\} \subseteq \textsc{Pos}(\varphi_1, \ldots, \varphi_n)$ is maximal.

Finally for a sequence $(\varphi_1, \ldots, \varphi_n)$ consisting of n (not necessarily distinct) literals from $\mathcal{L}_t(\text{sig})$ we define

$$\textsc{Set}(\varphi_1, \ldots, \varphi_n) = \bigcup_{i=1}^{n} \{\varphi_i\}.$$

Using these definitions the extension of the refinement operator for first order logic clauses to Prolog(+T)–rules is straightforward. As in the case of downward refinement we will work on sets of literals instead of rules.

$$\begin{aligned}
\Theta^{\mathcal{R}}_u(S_C) \;=\; & \left\{ S_C[\mathtt{Z}]_{p_1,\ldots,p_k} \;\middle|\; \begin{array}{r} t \in \textsc{Terms}(C), t = \mathtt{f}\left(\mathtt{X}_1, \ldots, \mathtt{X}_{\alpha(\mathtt{f})}\right) \text{ simple} \\ \{p_1, \ldots, p_k\} = \textsc{Occ}(t, C) \text{ and every } p \in \textsc{Occ}(\mathtt{X}_i, C) \\ \text{is inside of one } p_i \text{ for } i = 1, \ldots, \alpha(\mathtt{f}), \mathtt{Z} \notin \textsc{Var}(\mathrm{C}) \end{array} \right\} \\
\cup\; & \left\{ \textsc{Set}(dup(S_C, \mathtt{a})[\mathtt{Z}]_{p_1,\ldots,p_k}) \;\middle|\; \begin{array}{r} \mathtt{a} \in F, \alpha(\mathtt{a}) = 0, \mathtt{Z} \notin \textsc{Var}(C) \\ \emptyset \neq \{p_1, \ldots, p_k\} \subseteq \textsc{Occ}(\mathtt{a}, dup(S_C, \mathtt{a})) \end{array} \right\} \\
\cup\; & \left\{ \textsc{Set}(dup(S_C, \mathtt{X})[\mathtt{Z}]_{p_1,\ldots,p_k}) \;\middle|\; \begin{array}{r} \mathtt{X} \in \textsc{Var}(C), \mathtt{Z} \notin \textsc{Var}(C) \\ \emptyset \neq \{p_1, \ldots, p_k\} \subset \textsc{Occ}(\mathtt{X}, dup(S_C, \mathtt{X})), \end{array} \right\} \\
\cup\; & \{S_C \setminus \{\varphi\} \mid \varphi \in S_C \text{ is most general wrt. } C\}.
\end{aligned}$$

Given C we can therefore compute S_C and then apply $\Theta^{\mathcal{R}}_u$ to C. From the resulting set the extraction of a set of generalized rules is then carried out as usual. The procedure for computing the set of generalized rules is summarized in Algorithm 10.

Theorem 7.2.2

$\Theta^{\mathcal{R}}_u$ is locally finite and complete.

Algorithm 10 Upward–Refinement of PROLOG(+T)–rules

```
Input: PROLOG(+T)–rule $C = \varphi :- \psi_1, \ldots, \psi_n.$
Output: set of generalized rules
Require: $C = \varphi :- \psi_1, \ldots, \psi_n.$
 1: $S_C \leftarrow \{\varphi, \neg\psi_1, \ldots, \neg\psi_n\}$
 2: $Ref \leftarrow \emptyset$ {set of sets of literals}
Require: $S_C = \{\chi_0, \ldots, \chi_n\}$, $\mathtt{Z} \notin \textsc{Var}(C)$
Require: $F = \{\mathtt{f}_1, \ldots, \mathtt{f}_{|F|}\}$, $\textsc{Terms}(C) = \{t_1, \ldots, t_m\}$
 3: for $i = 1, \ldots, m$ do
 4:   if $t_i$ is simple then
Require:     $\textsc{Var}(t_i) = \{\mathtt{Z}_1, \ldots, \mathtt{Z}_l\}$
 5:     for $o = 1, \ldots, l$ do
 6:       if each $p \in \textsc{Occ}(\mathtt{X}_o, C)$ is inside a $p' \in \textsc{Occ}(t_i, C)$ then
Require:         $\{p_1, \ldots, p_k\}\textsc{Occ}(t_i, C)$
 7:         $Ref \leftarrow Ref \cup \{S_C[\mathtt{Z}]_{p_1,\ldots,p_k}\}$
 8:       end if
 9:     end for
10:   end if
11: end for
12: for $i = 1, \ldots, |F|$ do
13:   if $\alpha(\mathtt{f}_i) = 0$ then
14:     for each $\emptyset \neq \{p_1, \ldots, p_k\} \subseteq \textsc{Occ}(\mathtt{f}_i, dup(S_C, \mathtt{f}_i))$ do
15:       $Ref \leftarrow Ref \cup \{\textsc{Set}(dup(S_C, \mathtt{f}_i)[\mathtt{Z}]_{p_1,\ldots,p_k})\}$
16:     end for
17:   end if
18: end for
Require: $\textsc{Var}(C) = \{\mathtt{X}_1, \ldots, \mathtt{X}_l\}$
19: for $i = 1, \ldots, l$ do
20:   for each $\emptyset \neq \{p_1, \ldots, p_k\} \subset \textsc{Occ}(\mathtt{X}_i, dup(S_C, \mathtt{X}_i))$ do
21:     $Ref \leftarrow Ref \cup \{\textsc{Set}(dup(S_C, \mathtt{X}_i)[\mathtt{Z}]_{p_1,\ldots,p_k})\}$
22:   end for
23: end for
24: for $i = 0, \ldots, n$ do
25:   if $\chi_i$ is most general wrt. $C$ then
26:     $Ref \leftarrow Ref \cup \{S_C \setminus \{\chi_i\}\}$
27:   end if
28: end for
Require: $Ref = \{S_1, \ldots, S_{|S|}\}$
29: $R \leftarrow \emptyset$ {set of rules}
30: for $i = 1, \ldots, |S|$ do
Require:   $S_i = \left\{l_1^{(i)}, \ldots, l_{n_i}^{(i)}\right\}$
31:   for $j = 1, \ldots, n_i$ do
32:     if $l_j$ is positive then
33:       $R \leftarrow R \cup \{l_j :- \textsc{Tail}(S_i \setminus \{l_j\}).\}$
34:     end if
35:   end for
36: end for
37: return $R$
```

Proof. As for the operator $\Theta_d^{\mathcal{R}}$, locally finiteness is immediately clear from the definition of $\Theta_u^{\mathcal{R}}$. For the completeness the argumentation from [126] can be directly adapted to $\Theta_u^{\mathcal{R}}$. So the theorem is proved. $\square$

7.3. Refinement Strategies

Up to now we have defined operators for refining Prolog(+T)–literals and –rules both upward and downward. However, it is not yet clear how these operators should be applied. For example, exhaustive application of $\Theta_d^{\mathcal{R}}$ to any clause is not applicable as we have seen in Example 7.2.1. The size of the set computed by Algorithm 9 is given as stated in the following lemma[2].

Lemma 7.3.1

Let C be a Prolog(+T)–rule built over sig $= (\mathcal{X}, F, P, \alpha)$. Then

$$|\Theta_d^{\mathcal{R}}(C)| \leq |\textsc{Var}(C)|\,(|F| + |\textsc{Var}(C)| - 1) + 6|S_C| + 4|S_C|^2 + 2|P|.$$

Proof. We have

$$\begin{aligned}
|\Theta_d^{\mathcal{R}}(C)| \;\leq\; & \left|\left\{ S_C\left\{\frac{\mathtt{Z}}{\mathtt{f(X_1,\ldots,X_l)}}\right\} \mid \begin{array}{c} \mathtt{Z} \in \textsc{Var}(C), \mathtt{f} \in F, \alpha(\mathtt{f}) = l, \\ \mathtt{X}_1,\ldots,\mathtt{X}_l \notin \textsc{Var}(C), \\ \mathtt{X}_i \neq \mathtt{X}_j \text{ for } i \neq j \end{array}\right\}\right| \\
& + \left|\left\{ S_C\left\{\frac{\mathtt{Z}}{\mathtt{X}}\right\} \mid \mathtt{X},\mathtt{Z} \in \textsc{Var}(C), \mathtt{X} \neq \mathtt{Z}\right\}\right| \\
& + \left|\left\{ S_C \cup \{\mathtt{p(X_1,\ldots,X_l)}\} \mid \begin{array}{c} \mathtt{p} \in P, \alpha(\mathtt{p}) = l, \mathtt{X}_1,\ldots,\mathtt{X}_l \notin \textsc{Var}(C) \\ \mathtt{X}_i \neq \mathtt{X}_j \text{ for } i \neq j \end{array}\right\}\right| \\
& + \left|\left\{ S_C \cup \{\mathtt{not(p(X_1,\ldots,X_l))}\} \mid \begin{array}{c} \mathtt{p} \in P, \alpha(\mathtt{p}) = l, \mathtt{X}_1,\ldots,\mathtt{X}_l \notin \textsc{Var}(C) \\ \mathtt{X}_i \neq \mathtt{X}_j \text{ for } i \neq j \end{array}\right\}\right|
\end{aligned}$$

[2]We will restrict ourselves on the case of downward refinement since upward refinement often yields smaller sets of rules.

$$+|\{S_C \cup \{\mathsf{X}\psi\} \mid \psi \in S_C\}| + |\{S_C \cup \{\mathtt{not}(\mathsf{X}\psi)\} \mid \psi \in S_C\}|$$
$$+|\{S_C \cup \{\mathsf{G}\psi\} \mid \psi \in S_C\}| + |\{S_C \cup \{\mathtt{not}(\mathsf{G}\psi)\} \mid \psi \in S_C\}|$$
$$+|\{S_C \cup \{\mathsf{F}\psi\} \mid \psi \in S_C\}| + |\{S_C \cup \{\mathtt{not}(\mathsf{F}\psi)\} \mid \psi \in S_C\}|$$
$$+|\{S_C \cup \{\psi_1 \mathsf{U} \psi_2\} \mid \psi_1, \psi_2 \in S_C\}|$$
$$+|\{S_C \cup \{\neg\psi_1 \mathsf{U} \psi_2\} \mid \psi_1, \psi_2 \in S_C\}|$$
$$+|\{S_C \cup \{\psi_1 \mathsf{P} \psi_2\} \mid \psi_1, \psi_2 \in S_C\}|$$
$$+|\{S_C \cup \{\neg\psi_1 \mathsf{P} \psi_2\} \mid \psi_1, \psi_2 \in S_C\}|$$

by definition of $\Theta_d^{\mathcal{R}}$. Furthermore

- $$\left|\left\{ S_C \left\{ \frac{\mathtt{Z}}{\mathtt{f}(\mathtt{X}_1, \ldots, \mathtt{X}_l)} \right\} \mid \begin{array}{c} \mathtt{Z} \in \textsc{Var}(C), \mathtt{f} \in F, \alpha(\mathtt{f}) = l, \\ \mathtt{X}_1, \ldots, \mathtt{X}_l \notin \textsc{Var}(C), \\ \mathtt{X}_i \neq \mathtt{X}_j \text{ for } i \neq j \end{array} \right\}\right| \leq |\textsc{Var}(C)| \cdot |F|,$$

- $$\left|\left\{ S_C \left\{ \frac{\mathtt{Z}}{\mathtt{X}} \right\} \mid \mathtt{X}, \mathtt{Z} \in \textsc{Var}(C), \mathtt{X} \neq \mathtt{Z} \right\}\right| \leq |\textsc{Var}(C)| \cdot (|\textsc{Var}(C)| - 1),$$

- $$\left|\left\{ S_C \cup \{\mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_l)\} \mid \begin{array}{c} \mathtt{p} \in P, \alpha(\mathtt{p}) = l, \mathtt{X}_1, \ldots, \mathtt{X}_l \notin \textsc{Var}(C) \\ \mathtt{X}_i \neq \mathtt{X}_j \text{ for } i \neq j \end{array} \right\}\right| \leq |P| \text{ and}$$
$$\left|\left\{ S_C \cup \{\mathtt{not}(\mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_l))\} \mid \begin{array}{c} \mathtt{p} \in P, \alpha(\mathtt{p}) = l, \mathtt{X}_1, \ldots, \mathtt{X}_l \notin \textsc{Var}(C) \\ \mathtt{X}_i \neq \mathtt{X}_j \text{ for } i \neq j \end{array} \right\}\right| \leq |P|,$$

- $$|\{S_C \cup \{\mathsf{X}\psi\} \mid \psi \in S_C\}| \leq |S_C|,$$

$$\begin{aligned}
|\{S_C \cup \{\mathtt{not(X}\psi\mathtt{)}\} \mid \psi \in S_C\}| &\leq |S_C|, \\
|\{S_C \cup \{\mathsf{G}\psi\} \mid \psi \in S_C\}| &\leq |S_C|, \\
|\{S_C \cup \{\mathtt{not(G}\psi\mathtt{)}\} \mid \psi \in S_C\}| &\leq |S_C|, \\
|\{S_C \cup \{\mathsf{F}\psi\} \mid \psi \in S_C\}| &\leq |S_C| \text{ and} \\
|\{S_C \cup \{\mathtt{not(F}\psi\mathtt{)}\} \mid \psi \in S_C\}| &\leq |S_C|,
\end{aligned}$$

- $$\begin{aligned}
|\{S_C \cup \{\psi_1 \mathsf{U} \psi_2\} \mid \psi_1, \psi_2 \in S_C\}| &\leq |S_C|^2, \\
|\{S_C \cup \{\mathtt{not(}\psi_1 \mathsf{U} \psi_2\mathtt{)}\} \mid \psi_1, \psi_2 \in S_C\}| &\leq |S_C|^2, \\
|\{S_C \cup \{\psi_1 \mathsf{P} \psi_2\} \mid \psi_1, \psi_2 \in S_C\}| &\leq |S_C|^2 \text{ and} \\
|\{S_C \cup \{\mathtt{not(}\psi_1 \mathsf{P} \psi_2\mathtt{)}\} \mid \psi_1, \psi_2 \in S_C\}| &\leq |S_C|^2.
\end{aligned}$$

Combining these inequalities we have

$$\begin{aligned}
|\Theta_d^{\mathcal{R}}(C)| &\leq |\textsc{Var}(C)| \cdot |F| + |\textsc{Var}(C)| \cdot (|\textsc{Var}(C)| - 1) + 6|S_C| + 4|S_C|^2 + 2|P| \\
&= |\textsc{Var}(C)| \cdot (|F| + |\textsc{Var}(C)| - 1) + 6|S_C| + 4|S_C|^2 + 2|P|
\end{aligned}$$

as claimed. □

We will now see how refinement steps can be carried out *without* constructing the maximum number of rules.

7.3.1. Elimination of Variants

Recall that two PROLOG(+T)–rules C_1 and C_2 are variants if there are substitutions θ_1 and θ_2 such that $\theta_1(C_1) = C_2$ and $\theta_2(C_2) = C_1$. Since in our setting all rules in programs are considered universally closed, only one (of possibly arbitrary many) variants must be included in the set of refined rules. In the example from the last section, one of the rules 7 and 8 may be dropped.

In general, the presence of variants in the set of refinements is due to the construction of the loop from line 8 to line 14 in Algorithm 9. If we change the j–loop to range from $i+1$ to k and drop the **if**–condition (which now has no effect), the variants are not added.

In the original algorithm the number of rules added in the loop is given as $k^2 - k$. If the modification is added, only $\sum_{i=1}^{k}\sum_{j=i+1}^{k} 1$ rules are added. So the difference in the size of the *original* set of refined rules and the *modified* set is given as

$$\begin{aligned} k^2 - k - \left(k^2 - \frac{k(k+1)}{2}\right) &= \frac{k(k+1)}{2} - k \\ &= \frac{k^2 + k - 2k}{2} \\ &= \frac{k^2 - k}{2}. \end{aligned}$$

Setting $k = |\mathrm{VAR}(C)|$ and using Lemma 7.3.1 we have

$$\begin{aligned} \left|\Theta_d^{\mathcal{R}}(C)\right| &\leq |\mathrm{VAR}(C)| \cdot (|F| + |\mathrm{VAR}(C)| - 1) + 6|S_C| + 4|S_C|^2 \\ &\quad +2|P| - \frac{|\mathrm{VAR}(C)|^2 - |\mathrm{VAR}(C)|}{2} \\ &= |\mathrm{VAR}(C)| \cdot \left(|F| + \frac{|\mathrm{VAR}(C)|}{2} - \frac{1}{2}\right) + 6|S_C| + 4|S_C|^2 + 2|P|, \end{aligned}$$

where $\Theta_d^{\mathcal{R}}$ from now on denotes the refinement operator which implements the above strategy.

The growth of the set of refinements (depending on the number of variables in the rule

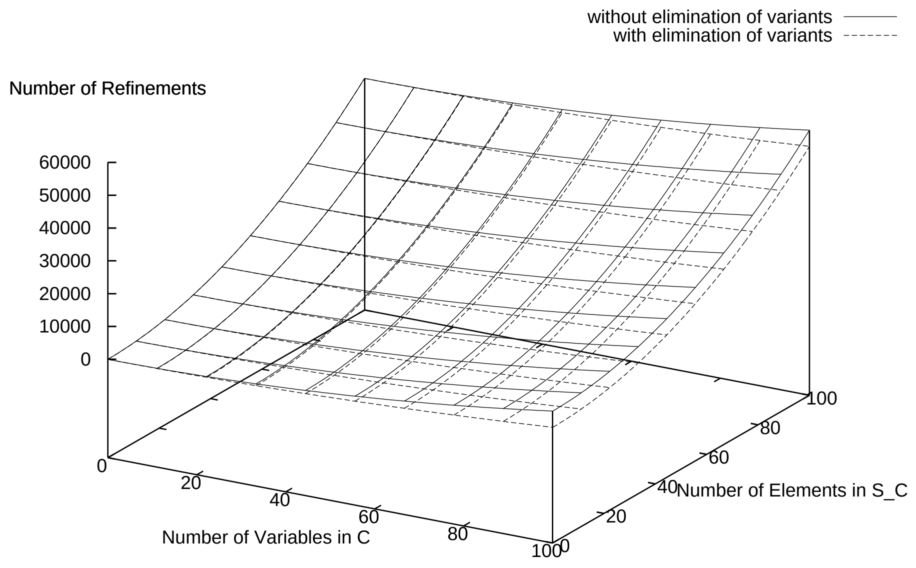

Figure 7.1.: Growth rate of refined rules with and without elimination of variants

to be refined and the size of the original rule) is depicted in Figure 7.1 using $|F| = 3$ and $|P| = 2$.

7.3.2. Restriction to reduced Rules

We have introduced the concept of ***reduced literals*** in order to keep the representation of a literal canonical, so that we can assume that each literal from $\mathcal{L}_t(\text{sig})$ has a certain form. Similarly a rule is reduced if every literal in this rule is reduced. Restricting ourselves to the construction of reduced rules during the refinement of Prolog(+T)–rules guarantees canonicity.

Example 7.3.1

Again consider rule

$$C = \mathsf{Gp}(\mathtt{X}_1) \text{ :–} \mathsf{Fq}(\mathtt{X}_1, \mathtt{f}(\mathtt{X}_2)), \mathsf{q}(\mathsf{a}, \mathsf{a})\mathsf{Up}(\mathsf{a}).$$

Then the set of refinements contains among others the rules

$$\begin{aligned} C_1 &= \mathtt{Gp(X_1)} :\!-\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a), not(GGp(X_1))}. \text{ and} \\ C_2 &= \mathtt{GGp(X_1)} :\!-\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a), not(Gp(X_1))}. \end{aligned}$$

These rules are syntactically different. However, reduction of the literals involved yields:

$$\begin{aligned} \textsc{Red}(C_1) &= \mathtt{Gp(X_1)} :\!-\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a), not(Gp(X_1))}. \text{ and} \\ \textsc{Red}(C_2) &= \mathtt{Gp(X_1)} :\!-\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a), not(Gp(X_1))}. \end{aligned}$$

Since $\textsc{Red}(C_1) = \textsc{Red}(C_2)$ one of the refinement steps can be skipped.

But the example from above yields even more possible improvements: both rules can be skipped. This is simply due to

$$\begin{aligned} \textsc{Red}(C_1) &= \mathtt{Gp(X_1)} :\!-\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a), not(Gp(X_1))}. \\ &\equiv \mathtt{Gp(X_1)} \vee \mathtt{not(Fq(X_1, f(X_2)))} \vee \mathtt{not(q(a,a))Up(a)} \vee \mathtt{not(not(Gp(X_1)))} \\ &\equiv \mathtt{Gp(X_1)} \vee \mathtt{not(Fq(X_1, f(X_2)))} \vee \mathtt{q(a,a)Up(a)} \\ &\equiv \mathtt{Gp(X_1)} :\!-\mathtt{Fq(X_1, f(X_2)), q(a,a)Up(a)}. \\ &= C. \end{aligned}$$

However, estimating the number of rules which can be skipped by restriction to reduced literals depends on the structure of the literals involved in the refined rule. So in general we are not able to give an estimation of the reduction of $|\Theta_d^{\mathcal{R}}(C)|$.

7.3.3. Elimination of Tautologies

Tautologies can be considered as rules which do not have any effect on the provability resp. non–provability of goals. Therefore they do not have to be constructed. A sufficient criterion is stated in the following lemma.

Lemma 7.3.2

Let C be any PROLOG(+T)–rule built over the signature sig. If there is $\varphi \in \mathcal{L}_t(\text{sig})$ such that $\{\varphi, \texttt{not}(\varphi)\} \subseteq S_C$, then C is a tautology.

Proof. immediately from

$$\begin{aligned} C &\equiv S_C \\ &= \{\varphi, \texttt{not}(\varphi), \psi_1, \ldots, \psi_k\} \text{ for } \varphi, \psi_1, \ldots, \psi_k \in \mathcal{L}_t(\text{sig}) \\ &\equiv \varphi \vee \texttt{not}(\varphi) \vee \psi_1 \vee \cdots \vee \psi_k \\ &\equiv \texttt{true} \vee \psi_1 \cdots \vee \psi_k \\ &\equiv \texttt{true}. \end{aligned}$$

□

Example 7.3.2

Let the rule

$$C = \texttt{Gp(X)} :- \texttt{p(X)}, \texttt{Fq(X,X)}.$$

be given. Then $\{\texttt{Gp(X)} :- \texttt{p(X)}, \texttt{Fq(X,X)}, \texttt{Gp(X)}\} \subseteq \Theta_d^{\mathcal{R}}(C)$ which is a tautology and can therefore be skipped.

In general, testing a rule C for being a tautology using the approach sketched above can be accomplished in time $\mathcal{O}\left(|S_C|^2\right)$. But similarly as in the foregoing section we cannot estimate the number of rules which might be skipped by tautology–elimination without taking the structure of C into account.

7.3.4. Premises vs. Conclusions

The downward refinement operator $\Theta_d^{\mathcal{R}}$ adds positive as well as negative literals to the set of refinements. Of course, every rule which is generated in this way is a downward refinement of the original rule. But one can rely on the following point of view: *the conclusions of rules should be known in advance, therefore it is better to adjust the premises.*

In the definition of $\Theta_d^{\mathcal{R}}$ this is modeled by only adding negative literals during the execution of the loop in lines 15–22. The number of literals added by only adding negative literals is then

$$|P| + 3|S_C| + 2|S_C|^2,$$

so the overall number of refinements is

$$\left|\Theta_d^{\mathcal{R}}(C)\right| \leq |\textsc{Var}(C)| \cdot \left(|F| + \frac{|\textsc{Var}(C)|}{2} - \frac{1}{2}\right) + 3|S_C| + 2|S_C|^2 + |P|.$$

Figure 7.2 compares the introduced strategy with the strategy of elimination of variants described above.

However, by adding negative literals only we lose the completeness of the operator. This can be easily seen as follows: consider the rules

$$\begin{aligned} C_1 &= \mathsf{Gp}(\mathtt{X}_1) :- \mathsf{Fq}(\mathtt{X}_1, \mathtt{f}(\mathtt{X}_2)), \mathtt{q}(\mathtt{a}, \mathtt{a}) \mathsf{Up}(\mathtt{a}). \text{ and} \\ C_2 &= \mathsf{Gp}(\mathtt{X}_1) :- \mathsf{Fq}(\mathtt{X}_1, \mathtt{f}(\mathtt{X}_2)), \mathtt{q}(\mathtt{a}, \mathtt{a}) \mathsf{Up}(\mathtt{a}), \mathtt{not}(\mathtt{q}(\mathtt{X}_1, \mathtt{X}_1)). \end{aligned}$$

Since only negative literals are added in the process of refining, every rule C contained in $\left(\Theta_d^{\mathcal{R}}\right)^n(C)$ for any n only may contain other *positive* premises. So no rule $C' \approx_s C_2$ will be constructed.

The above results regarding techniques for the restriction of the number of refinements complete our study of refinement operators for Prolog(+T)–objects. We have seen that refinement operators for nontemporal logic programming languages can be natu-

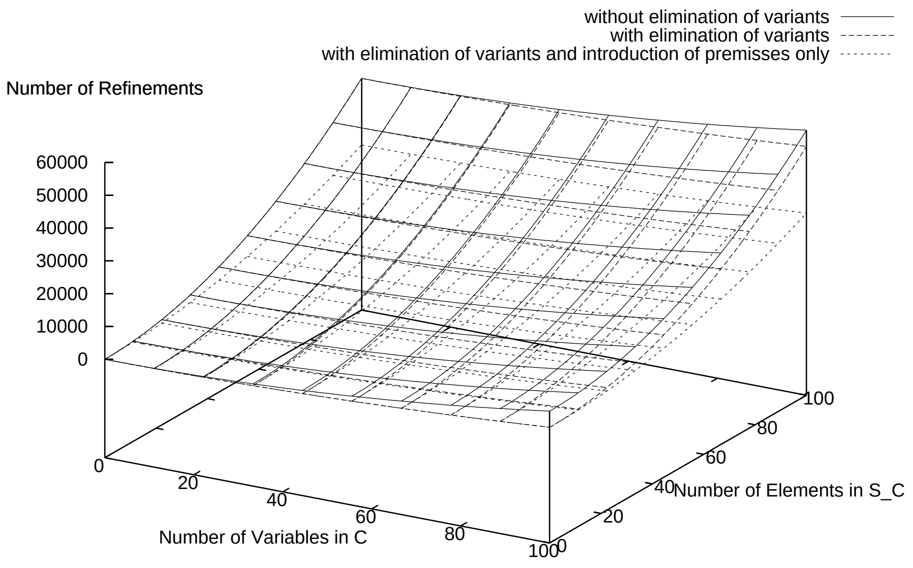

Figure 7.2.: Growth Rate by adding Premises only

rally generalized in order to include mechanisms for refining objects containing temporal operators. Therefore refinement operators for PROLOG(+T)–objects could be easily derived by adapting several well–known techniques from the field of first order logic programming.

What remains to be studied is the complexity of the search for a correct program given sets $\mathcal{E}^+$ and $\mathcal{E}^-$ of examples. This topic will be attacked in the following chapter.

8. Identifiability of Prolog(+T)–programs

Contents

8.1. PAC–Learning **170**

8.2. Learnability **177**

8.2.1. The general case 177

8.2.2. Programs with syntactical restrictions 183

The operators from the last chapters always create programs which are at every point of time correct with respect to the examples which have been presented so far. But in order to be of great use it is necessary to be able to ensure that *other examples*, that is examples which have not (yet) been seen are classified correct. Consider two sets $\mathcal{E}^+$ and $\mathcal{E}^-$ of examples and a program P constructed by the algorithms from the last chapters giving these examples as inputs. Furthermore assume that e is any ground atom from $B_P^{\text{FoLtl}} \setminus (\mathcal{E}^+ \cup \mathcal{E}^-)$. In order to be *good*, P should classify e correct. Formally let P_{cor} be the *correct* program from which the examples from $\mathcal{E}^+$ and $\mathcal{E}^-$ are derived, that is let P_{cor} be the program to be identified. Then we want P to have the following properties:

- if $P_{\text{cor}} \models e$, then $P \models e$ and
- if $P_{\text{cor}} \not\models e$, then $P \not\models e$.

In other words

$$\{e \mid P_{\text{cor}} \models e\} = \{e \mid \ P \models e\} .$$

In general this cannot be reached. Therefore we will adapt a model of *learning* which allows some errors with a certain probability. The issue of identifying logic programs in the PAC–setting has been studied in depth in [36] and [37]. In general not every program is identifiable. But for certain subsets identifiability can be ensured.

8.1. PAC–Learning

PAC–learning is a model of learning which has been introduced by Vailant (see [157]). Any algorithm which has to solve the learning problem is evaluated with respect to two parameters ε and δ which specify a limit for the *difference* between the concept to be learned and the actual hypothesis (ε) and the probability that the actual difference is greater than this level (δ).

In order to give formal definitions of PAC–learnability we need some more concepts. Single objects to be identified will be referred to as *concepts*, the sets of all such objects will be called *concept classes*. Each concept class is defined over some set X. Formally a concept class over X is a set $\mathcal{C} \subseteq 2^X$. So a concept is an element $C \in \mathcal{C}$, that is a set $C \in 2^X$.

In our case of learning PROLOG(+T)–programs we assume that a signature sig $= (\mathcal{X}, F, P, \alpha)$ is given. Then we define X to consists of all sets of PROLOG(+T)–literals which contain at least one positive literal, that is

$$X := \{C \subseteq \mathcal{L}_t(\text{sig}) \mid |\textsc{Pos}(C)| \geq 1\} .$$

Concepts are PROLOG(+T)–programs, that is sets $P \subseteq X$ and concept classes are sets of PROLOG(+T)–programs, that is sets $\mathcal{C} \subseteq 2^X$.

In order to identify a particular concept an algorithm has to process some kind of examples. In [157] and [63] the examples are considered to be elements from X. However, it will be more adequate for us to consider only ground atoms as examples. So an *example* is an element φ from the set $\{\psi \in \mathcal{A}_t(\text{sig}) \mid \textsc{Var}(\varphi) = \emptyset\}$. A *classified example* for a concept P is a tuple $\langle \varphi, v \rangle$ consisting of an example φ and $v \in \{0, 1\}$. If $v = 1$ we call the example *positive* and assume that $P \models \varphi$ and in the case that $v = 0$ we call it *negative* and therefore assume that $P \not\models \varphi$. For the sake of readability we will from now on use the following notational convention: let P be a Prolog(+T)–program and let φ be an example. Then

$$\begin{aligned} P(\varphi) = 1 &\quad\Leftrightarrow\quad P \models \varphi \text{ and} \\ P(\varphi) = 0 &\quad\Leftrightarrow\quad P \not\models \varphi. \end{aligned}$$

A *sample* for P is a sequence $S_P = (\langle \varphi_1, P(\varphi_1)\rangle, \ldots, \langle \varphi_n, P(\varphi_n)\rangle)$ of classified examples. The number n will be called the *length* of the sample or the *sample complexity*.

Let P and P' be Prolog(+T)–programs and let $S_P = (\langle \varphi_1, P(\varphi_1)\rangle, \ldots, \langle \varphi_n, P(\varphi_n)\rangle)$ be a sample of length n for P. P' is called *consistent* with respect to S_P or S_P*–consistent* if for every i it holds that $P'(\varphi_i) = P(\varphi_i)$, that is a consistent program classifies every example exactly in the same way as the program from which the examples are derived does.

We assume that the examples which are presented to the algorithms are chosen with respect to a fixed probability distribution D on the set of all ground atoms. Also we will sometimes write $\mathbf{Pr}_D$ instead of D or simply $\mathbf{Pr}$ if D is clear from the context.

Now let $\mathcal{S}(X, \mathcal{C})$ be the set of all samples which can be constructed for concepts from $\mathcal{C}$ if the examples are chosen with respect to D and let $\mathcal{C}$ and $\mathcal{H}$ be concept classes. A *learning algorithm* (or a *learner*) is a total function[1] $\mathcal{A}_{\mathcal{C},\mathcal{H}} : \mathcal{S}(X, \mathcal{C}) \to \mathcal{H}$. That is a

[1]That is, a function which is defined for all possible inputs.

learning algorithm is presented a sample of some length and constructs a *hypothesis* from the example which it has seen by processing the sample. In our case we will have $\mathcal{H} = \mathcal{C}$, so we do not have to distinguish between different representation languages.

As one might expect, a hypothesis computed by some learning algorithm $\mathcal{A}_{\mathcal{C},\mathcal{H}}$ is called *consistent with respect to a sample* $S_P = (\langle \varphi_i, P(\varphi_i)\rangle)_{i=1}^n$ if for every $i \in \{1, \ldots, n\}$ it holds that

$$\left(\mathcal{A}_{\mathcal{C},\mathcal{H}}(S_P)\right)(\varphi_i) = P(\varphi_i).$$

$\mathcal{A}_{\mathcal{C},\mathcal{H}}$ is called *consistent* if every hypothesis computed from a given sample is consistent with respect to this sample.

The *quality* of a hypothesis is measured by analyzing the probability that a randomly chosen example is contained in the symmetric difference between the *correct concept* and the *computed concept*. So the *error* of a program P' with respect to a program P is given as

$$\operatorname{error}(P, P') = D\left(P \Delta P'\right) = D\left((P \setminus P') \cup (P' \setminus P)\right).$$

The error of a hypothesis computed by a learning algorithm is usually given as a parameter called ε and one is interested in upper bounds for the probability that a learning algorithm or a classifier induces hypotheses such that the error of these hypotheses exceeds the value of ε. Several attempts have been carried out in order to analyze such errors. In 1971 Vapnik and Chervonenkis have shown (see [166]) that classifiers can be constructed which have an error ration bounded from above by $4s(\mathcal{C}, 2n)e^{-\frac{n\varepsilon^2}{8}}$ for a given value or ε where $s(\mathcal{C}, 2n)$ denotes the relative amount of samples of size $2n$ which can be drawn from the concept class $\mathcal{C}$. Devroye and Wagner (see [46]) have extended the results from [166] in order to derive distribution–free upper bounds for the error in the case of *half–planes*. They show that classifiers can be constructed with an error rate bounded by $4(1+2^d n_i^d)e^{-\frac{n_i\varepsilon^2}{8}}$ for $i = 1, 2$ where d denotes the size of the samples under consideration while the problem domain is divided into classes $\mathcal{C}_1$ and $\mathcal{C}_2$. Furthermore they extend their results to concept classes with higher dimension yielding similar results.

Furthermore some attempts for analyzing the learning task in the domain of identifying boolean functions have been carried out. In [131] Pearl derives upper and lower bounds for the size of the value $s(n, c)$ denoting the relative frequency of samples of size n derived from a concept class representing a boolean function which can be built up with at most c binary gates by proving that

$$s(n,c) \geq \begin{cases} 2^{n_0-n} & \text{if } n \geq n_0 \\ 1 & \text{else} \end{cases}$$

and

$$s(n,c) \leq 2^{\log_2 |F_c| - n}$$

where n_0 denotes the maximum value of n such that a presented sample e_n can be embedded in a boolean function f using at most c logical gates (where $|F_c|$ denotes the total number of boolean functions containing at most c gates). Using these bounds Pearl proves that classifiers can be constructed which have an error ration bounded from above by

$$\left(\sqrt{2\ln 2 \cdot c(2+\log_2 c)} + \frac{1}{\sqrt{2\ln 2 \cdot c(2+\log_2 c)}}\right)\frac{2}{\sqrt{n}}.$$

Finally Devroye (see [45]) derives upper bounds for the error of classifiers for both the case of finite and infinite concept classes. For finite classes $\mathcal{C}$ he proves that the error can be bounded from above by $2|\mathcal{C}|e^{-2m\varepsilon^2}$ (using samples of size m) and that the expected error is given by

$$\sqrt{\frac{\log(2|\mathcal{C}|}{2m}} + \frac{1}{\sqrt{8m\log(2|\mathcal{C}|)}}.$$

For infinite concept classes they show that the error can be bounded from above by

$cs(\mathcal{C}, m^2)e^{-2m\varepsilon^2}$ and that the expected error is given by

$$\sqrt{\frac{\log(4e^8 s(\mathcal{C}, m^2))}{2m}} + \frac{1}{\sqrt{8m\log(4e^8 s(\mathcal{C}, m^2))}}.$$

Having presented all necessary prerequisites we can now formally define the concept of *PAC–identifiability*. For our definition we will only slightly change the notations from [22].

Definition 8.1.1 (PAC–Learnability, Valiant [157] and Blumer et al. [22]) Let X be some set and let $\mathcal{C}$ and $\mathcal{H}$ be concept classes over X. $\mathcal{C}$ is called *PAC–learnable* using $\mathcal{H}$ if and only if there is a learning algorithm $\mathcal{A}_{\mathcal{C},\mathcal{H}}$ and a function $m : \mathbb{R}^2 \to \mathbb{R}$ such that for every probability distribution D, every $P \in \mathcal{C}$ and every values of $0 < \varepsilon, \delta < 1$ it holds that:

1. $\mathcal{A}_{\mathcal{C},\mathcal{H}}$ is presented some sample S_P of length $\lceil m(\varepsilon, \delta) \rceil$,
2. $\mathcal{A}_{\mathcal{C},\mathcal{H}}(S_P)$ is put out and
3. $\mathbf{Pr}(\text{error}(\mathcal{A}_{\mathcal{C},\mathcal{H}}(S_C, P)) \geq \varepsilon) = \mathbf{Pr}\left(D\left((\mathcal{A}_{\mathcal{C},\mathcal{H}}(S_P))\,\Delta P\right) \geq \varepsilon\right) < \delta.$

The last point from the above definition is usually referred to as the *PAC–criterion* or the *PAC–criterion with respect to ε and δ.*

So $\mathcal{C}$ is PAC–learnable if there exists an algorithm which regardless of the underlying distribution (which determines *how* the examples are chosen) only needs to process a finite set of examples in order to keep the difference between the hypothesis and the correct program *small* ($\leq \varepsilon$) with a high probability ($< \delta$).

The PAC concept has been introduced in [157] for the domain of learning boolean functions. Often PAC–identifiability is referred to as an abbreviation to *polynomial time PAC–identifiability* where a further restriction is put on the learning algorithm $\mathcal{A}_{\mathcal{C},\mathcal{H}}$, namely that its runtime is bounded from above by some suitable polynomial. So in this case the term $\mathcal{C}$ *is PAC–identifiable* means that $\mathcal{C}$ *is polynomial time PAC–identifiable.* Consequently there might be concept classes which are PAC–identifiable in our concept which does not include this requirement for the runtime of learning algorithms but which are not polynomial time PAC–identifiable. In [158] and [132] L.G. Valiant addresses this problem deriving several classes of relatively simply structured boolean functions which are not PAC–identifiable if polynomial runtime is required by a learning algorithm.

In order to characterize the complexity of learning single concepts from a concept class $\mathcal{C}$ the *Vapnik–Chervonenkis–Dimension* has been proven to be an adequate parameter. Intuitively the Vapnik–Chervonenkis–Dimension characterizes the difficulty of how to distinguish between different objects from $\mathcal{C}$. This intuition will now be made formally clear.

Definition 8.1.2 (Blumer et al. [22])

Let $\mathcal{C}$ be a concept class over some set X and let $T \in \mathcal{C}$ be a concept. Then

$$\Pi_{\mathcal{C}}(T) = \{C \cap T \mid C \in \mathcal{C}\}.$$

A set T of cardinality k is said to be *shattered* by $\mathcal{C}$ if $|\Pi_{\mathcal{C}}(T)| = 2^k$, that is if $\Pi_{\mathcal{C}}(T) = 2^T$. So the sets C which are shattered by $\mathcal{C}$ can be seen as the most difficult concepts from $\mathcal{C}$. The Vapnik–Chervonenkis–Dimension of a concept class $\mathcal{C}$ is now defined to be the maximum size of a concept which is shattered by $\mathcal{C}$.

Definition 8.1.3 (VC–Dimension, Blumer et al. [22])

Let $\mathcal{C}$ be a concept class over some set X. The *Vapnik–Chervonenkis–Dimension* of $\mathcal{C}$ is defined as

$$\textsc{VCDim}(\mathcal{C}) \quad = \quad \max\left\{k \mid \text{ there is a } T \in \mathcal{C} \text{ with } |T| = k \text{ and } |\Pi_{\mathcal{C}}(T)| = 2^k\right\}$$

$$= \max\{|T| \mid T \in \mathcal{C} \text{ is shattered by } \mathcal{C}\}.$$

If no such k exists we will write $\mathrm{VCD{\scriptsize IM}}(\mathcal{C}) = \infty$. $\mathcal{C}$ is said to have *unbounded VC–Dimension* in this case.

The link between the VC–Dimension and the learnability of a concept class $\mathcal{C}$ is given by the following theorem (see [22]).

Theorem 8.1.1 (Blumer et al. [22])

Let $\mathcal{C}$ be a concept class over some set X. Then $\mathcal{C}$ is PAC–learnable if and only if $\mathrm{VCD{\scriptsize IM}}(\mathcal{C}) < \infty$.

The following theorem gives a possibility to estimate the length of the sample needed in order to identify the target concept given fixed values of ε and δ.

Theorem 8.1.2 (Blumer et al. [22])

Let $\mathcal{C}$ be a concept class over some set X such that $1 \le \mathrm{VCD{\scriptsize IM}}(\mathcal{C}) < \infty$ and let $0 < \varepsilon \le \frac{1}{2}$ and $0 < \delta < 1$ be given. Then every consistent learning algorithm for $\mathcal{C}$ using $\mathcal{C}$ needs to process at most

$$\left\lceil \max\left\{\frac{4}{\varepsilon}\ln\frac{4}{\delta},\ \ln\frac{8\mathrm{VCD{\scriptsize IM}}(\mathcal{C})}{\varepsilon}\ln\frac{13}{\varepsilon}\right\}\right\rceil$$

examples in order to ensure the PAC–criterion with respect to ε and δ[2].

In general estimating the VC–Dimension of some concept class $\mathcal{C}$ is a very difficult task. But in the case of finite concept classes, the VC–Dimension is bounded by the logarithm of the size of the class. This is the tenor of the following lemma.

Lemma 8.1.1 (Fischer [63])

Let $\mathcal{C}$ be a concept class over some set X. If $\mathcal{C}$ is finite, then

$$\mathrm{VCD{\scriptsize IM}}(\mathcal{C}) \le \log_2 |\mathcal{C}|.$$

[2]In general one has to take the VC–Dimension $\mathrm{VCD{\scriptsize IM}}(\mathcal{H})$ of the target concept class into account. But since we require $\mathcal{C} = \mathcal{H}$ this makes the analysis of the learning problem a bit easier.

In the following sections, Lemma 8.1.1 will be used in order to derive some upper bounds for the VC–Dimensions of several classes of Prolog(+T)–programs.

8.2. Learnability and Non–Learnability of selected classes of Prolog(+T)–programs

8.2.1. The general case

We will now derive upper and lower bounds for the VC–Dimension of some classes of Prolog(+T)–programs. Therefore we will extend some techniques presented recently in [11].

Definition 8.2.1

Let c, t, l and o be nonnegative integers. The class $\mathcal{P}^{\leq c,t,l,o}$ is defined as the set of all Prolog(+T)–programs P with the following properties:

1. P consists of at most c rules,
2. each rule in P consists of at most l literals,
3. each literal in a rule in P does not contain more than t distinct terms and
4. each literal in a rule in P does not contain more than o temporal operators.

Assume that a fixed signature sig $= (\mathcal{X}, F, P, \alpha)$ is given. We will from now on use the following abbreviations:

1. $f := |F|$,
2. $p := |P|$ and
3. $a := \max\{\alpha(\sigma) \mid \sigma \in F \cup P\}$.

Programs will be encoded as strings over the binary alphabet $\Sigma = \{0, 1\}$. We will see that for fixed values of c, t, l and o, the number $|\mathcal{P}^{\leq c,t,l,o}|$ is finite. The VC–Dimension can then be estimated using Lemma 8.1.1. We will assume that each literal in any program is reduced. This is no restriction since we have seen that each literal has a reduced normal form which can be effectively computed.

First we will review some of the results from [11]. There it is shown that a term containing a arguments and at most t distinct subterms can be encoded using $\log_2 f + a \log_2 t$ bits. Consequently a set of t (distinct) terms can be encoded using not more than $t(\log_2 f + a \log_2 t)$ bits. Since f and a are constant we have $t(\log_2 f + a \log_2 t) = \mathcal{O}(t \log_2 t)$.

We will now fix a numbering for the symbols from P and the temporal operators. Let P_{ext} denote the set $P \cup \{\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}, \mathsf{P}\}$ and assume that the set P is *ordered* as follows: $P = \{p_0, \ldots, p_{|P|-1}\}$. The symbols p_i will be mapped to $\text{bin}(i)$ where $\text{bin}(i)$ denotes the string representing the binary representation of i. Furthermore we fix the following mapping:

$$\begin{aligned}
\mathsf{X} &\mapsto \text{bin}(|P|+1), \\
\mathsf{F} &\mapsto \text{bin}(|P|+2), \\
\mathsf{G} &\mapsto \text{bin}(|P|+3), \\
\mathsf{U} &\mapsto \text{bin}(|P|+4) \text{ and} \\
\mathsf{P} &\mapsto \text{bin}(|P|+5),
\end{aligned}$$

where the strings might be padded with zeros on the left side in order to obtain strings of equal length.

Example 8.2.1

The set $P = \{p_0\}$ containing only a single predicate symbol yields the following mapping:

$$\begin{aligned} p_0 &\mapsto 000, \\ \mathsf{X} &\mapsto 001, \\ \mathsf{F} &\mapsto 010, \\ \mathsf{G} &\mapsto 011, \\ \mathsf{U} &\mapsto 100 \text{ and} \\ \mathsf{P} &\mapsto 101. \end{aligned}$$

Consequently any Prolog(+T)–literal containing at most t distinct terms and at most o temporal operators can be encoded using at most

$$o + 1 + 2\left(\lceil \log_2 (p+5) \rceil + \lceil \log_2 p \rceil + \lceil a \log_2 t \rceil\right)$$

bits. Therefore a rule consisting of at most l such literals can be encoded using at most

$$l\left(o + 1 + 2\left(\lceil \log_2 (p+5) \rceil + \lceil \log_2 p \rceil + \lceil a \log_2 t \rceil\right)\right)$$

bits and a program P containing at most c such rules can be encoded using at most

$$cl\left(o + 1 + 2\left(\lceil \log_2 (p+5) \rceil + \lceil \log_2 p \rceil + \lceil a \log_2 t \rceil\right)\right)$$

bits. This gives:

$$\begin{aligned} & cl\left(o + 1 + 2\left(\lceil \log_2 (p+5) \rceil + \lceil \log_2 p \rceil + \lceil a \log_2 t \rceil\right)\right) \\ = \;& clo + cl + 2cl\left(\lceil \log_2 (p+5) \rceil + \lceil \log_2 p \rceil + \lceil a \log_2 t \rceil\right) \end{aligned}$$

$$\begin{aligned}
&= cl(o+1) + 2cl\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil + \lceil a \log_2 t\rceil\right) \\
&= \mathcal{O}\left(cl(o+1) + 2cl\left(\log_2(p+5) + \log_2 p + a\log_2 t\right)\right) \\
&= \mathcal{O}\left(cl(o+1) + 2cl\left(\log_2\left((p^2+5p)t^a\right)\right)\right).
\end{aligned}$$

With this number of bits we can encode at most $2^{\mathcal{O}\left(cl(o+1)+2cl\left(\log_2\left((p^2+5p)t^a\right)\right)\right)}$ different bitstrings, that is we have

$$|\mathcal{P}^{\leq c,t,l,o}| = 2^{\mathcal{O}\left(cl(o+1)+2cl\left(\log_2\left((p^2+5p)t^a\right)\right)\right)} < \infty$$

and therefore

$$\textsc{VCDim}\left(\mathcal{P}^{\leq c,t,l,o}\right) = \mathcal{O}\left(cl(o+1) + 2cl\left(\log_2\left((p^2+5p)t^a\right)\right)\right)$$

using Lemma 8.1.1.

On the other hand the best case is given if no temporal operators are involved. [11] then gives the following estimation:

$$\textsc{VCDim}\left(\mathcal{P}^{\leq c,t,l,o}\right) = \Omega\left(cl + ct\right).$$

The results are summarized in the following theorem.

Theorem 8.2.1

Let c, t, l and o be fixed, nonnegative integers. Then

$$\begin{aligned}
\textsc{VCDim}\left(\mathcal{P}^{\leq c,t,l,o}\right) &= \mathcal{O}\left(c + (o+1) + 2cl\left(\log_2\left((p^2+5p)t^a\right)\right)\right) \text{ and} \\
\textsc{VCDim}\left(\mathcal{P}^{\leq c,t,l,o}\right) &= \Omega\left(cl + ct\right).
\end{aligned}$$

Using these equalities we can estimate the number of examples needed in order to ensure the PAC–criterion given fixed values of ε and δ.

Theorem 8.2.2

Let $c, t, l, o \geq 0$, $0 < \varepsilon \leq \frac{1}{2}$ and $0 < \delta < 1$ be fixed. Then every learning algorithm $\mathcal{A}$ needs at most

$$\max\left\{\frac{4}{\varepsilon}\ln\frac{4}{\delta}, \frac{8\mathcal{O}\left(cl(o+1) + 2cl\left(\log_2\left((p^2+5p)t^a\right)\right)\right)}{\varepsilon}\ln\frac{13}{\varepsilon}\right\}$$

examples in order to ensure the PAC–criterion.

Example 8.2.2

We conclude this section by illustrating the results for the number of examples. Let $o = 10$, $l = 30$, $c = 200$, $t = 50$, $p = 7$ and $a = 17$. Then the equation from Theorem 8.2.2 can be simplified to:

$$\begin{aligned}
& \max\left\{\frac{4}{\varepsilon}\ln\frac{4}{\delta}, \frac{8\text{VCD}\textsc{im}\left(\mathcal{P}^{\leq c,t,l,o}\right)}{\varepsilon}\ln\frac{13}{\varepsilon}\right\} \\
= \quad & \max\left\{\frac{4}{\varepsilon}\ln\frac{4}{\delta}, \frac{8\mathcal{O}\left(cl(o+1) + 2cl\left(\log_2\left((p^2+5p)t^a\right)\right)\right)}{\varepsilon}\ln\frac{13}{\varepsilon}\right\} \\
\approx \quad & \max\left\{\frac{4}{\varepsilon}\ln\frac{4}{\delta}, \frac{8\left(cl(o+1) + 2cl\left(\log_2\left((p^2+5p)t^a\right)\right)\right)}{\varepsilon}\ln\frac{13}{\varepsilon}\right\} \\
\approx \quad & \max\left\{\left\lceil\frac{4}{\varepsilon}\ln\frac{4}{\delta}\right\rceil, \left\lceil\frac{8\left(200\cdot 30(10+1) + 2\cdot 200\cdot 30\left(\log_2\left((7^2+5\cdot 7)50^{17}\right)\right)\right)}{\varepsilon}\ln\frac{13}{\varepsilon}\right\rceil\right\} \\
= \quad & \max\left\{\left\lceil\frac{4}{\varepsilon}\ln\frac{4}{\delta}\right\rceil, \left\lceil\frac{528000 + 96000\left(\log_2\left((7^2+5\cdot 7)50^{17}\right)\right)}{\varepsilon}\ln\frac{13}{\varepsilon}\right\rceil\right\} \\
\approx \quad & \max\left\{\left\lceil\frac{4}{\varepsilon}\ln\frac{4}{\delta}\right\rceil, \left\lceil\frac{528000 + 69000\cdot 96}{\varepsilon}\ln\frac{13}{\varepsilon}\right\rceil\right\} \\
= \quad & \max\left\{\left\lceil\frac{4}{\varepsilon}\ln\frac{4}{\delta}\right\rceil, \left\lceil\frac{9744000}{\varepsilon}\ln\frac{13}{\varepsilon}\right\rceil\right\}.
\end{aligned}$$

Note that the approximation given above is quite weak since the omission of the symbol $\mathcal{O}$ may result in omitting quite large constants.

Figure 8.1 illustrates the number of examples for c, t, l, o, p and a as above and variable values of ε and δ.

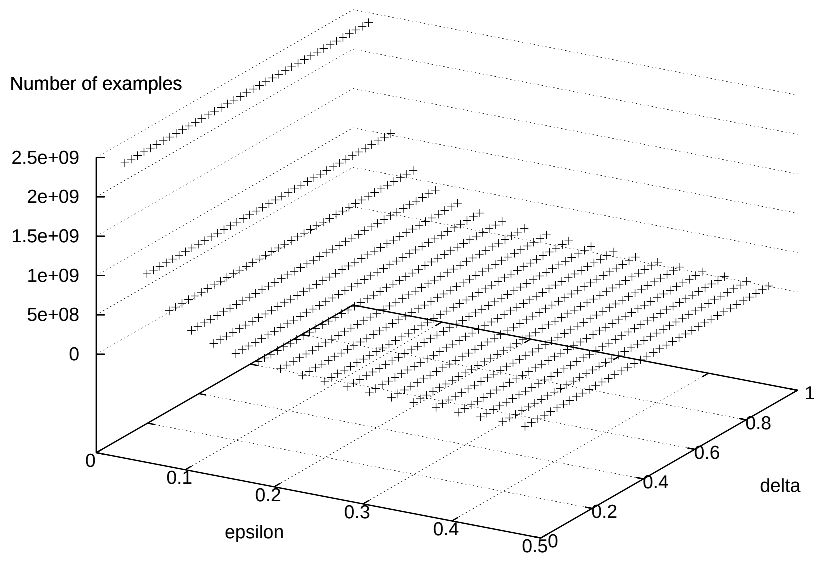

Figure 8.1.: Number of examples given fixed values for c, t, l, o, p and a with ε ranging from 0 to $\frac{1}{2}$ and δ ranging from 0 to 1.

8.2.2. Programs with syntactical restrictions

We will now see how restricting the form of the involved rules in programs can lower the VC–dimension of a class of PROLOG(+T)–programs and therefore make it easier to identify these programs by presenting positive and negative examples. Therefore we will study two classes of programs which have already been pointed out in [11]: *constrained* programs and *range–restricted* programs.

Definition 8.2.2 (Syntactical Restrictions, Arias and Khardon [11])
Let $C = \varphi \text{ :-} \psi_1, \ldots, \psi_n$. be a PROLOG(+T)–rule. C is called

- *range–restricted* if $\text{TERMS}(\varphi) \subseteq \bigcup_{i=1}^{n} \text{TERMS}(\psi_i)$ and
- *constrained* if $\bigcup_{i=1}^{n} \text{TERMS}(\psi_i) \subseteq \text{TERMS}(\varphi)$.

Consequently a PROLOG(+T)–program $P = \{P_1, \ldots, P_k\}$ is called *range–restricted* (resp. *constrained*) if every P_i is range–restricted (resp. constrained).

Fixing nonnegative integers c, t, l and o, the definition of the classes $\mathcal{P}_{\text{con}}^{\leq c,t,l,o}$ and $\mathcal{P}_{\text{rr}}^{\leq c,t,l,o}$ is as one might expect:

- $\mathcal{P}_{\text{con}}^{\leq c,t,l,o} = \left\{P \in \mathcal{P}^{\leq c,t,l,o} \mid P \text{ is constrained}\right\}$ and
- $\mathcal{P}_{\text{rr}}^{\leq c,t,l,o} = \left\{P \in \mathcal{P}^{\leq c,t,l,o} \mid P \text{ is range–restricted}\right\}$.

We will now study how the values of $\text{VCDIM}\left(\mathcal{P}_{\text{con}}^{\leq c,t,l,o}\right)$ and $\text{VCDIM}\left(\mathcal{P}_{\text{rr}}^{\leq c,t,l,o}\right)$ can be estimated using the results from the foregoing section.

The VC–Dimension of constrained Prolog(+T)–programs

Let $P \in \mathcal{P}_{\text{con}}^{\leq c,t,l,o}$ be given. Assume that $P = \{P_1, \ldots, P_k\}$ for some $k \leq c$ and

$$P_i = \varphi_i \text{ :-} \psi_1^{(i)}, \ldots, \psi_{n_i}^{(i)}.$$

Since $P \in \mathcal{P}_{\text{con}}^{\leq c,t,l,o}$ we have

$$\bigcup_{j=1}^{n_i} \textsc{Terms}\left(\psi_j^{(i)}\right) \subseteq \textsc{Terms}(\varphi_i)$$

for $i = 1, \ldots, k$. Therefore only the terms in the heads of the rules in P have to be encoded. This can be achieved by using at most $o{+}1{+}2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil + \lceil a \log_2 t\rceil\right)$ bits. The literals from $\left\{\psi_1^{(j)}, \ldots, \psi_{n_j}^{(j)}\right\}$ can be encoded by only encoding the negation symbols, the predicate symbol(s) and the temporal operators involved. Consequently this can be achieved by using at most

$$o + 1 + 2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right)$$

bits per literal. Therefore the tail of a rule containing at most l literals can be encoded by using at most

$$(l-1)\left(o + 1 + 2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right)\right)$$

bits. So a complete rule can be encoded using at most

$$o{+}1{+}2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil + \lceil a \log_2 t\rceil\right){+}(l{-}1)\left(o + 1 + 2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right)\right)$$

bits. We have

$$\begin{aligned}
& o + 1 + 2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil + \lceil a \log_2 t\rceil\right) \\
& +(l-1)\left(o + 1 + 2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right)\right) \\
= \; & o + 1 + 2\lceil \log_2(p+5)\rceil + 2\log_2 p\rceil + 2\lceil a \log_2 t\rceil + (l-1)(o+1) \\
& +2(l-1)\lceil \log_2(p+5)\rceil + 2(l-1)\lceil \log_2 p\rceil \\
= \; & l(o+1) + 2l\lceil \log_2(p+5)\rceil + 2l\lceil \log_2 p\rceil + 2\lceil a \log_2 t\rceil \\
= \; & l(o+1) + 2\left(l\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right) + \lceil a \log_2 t\rceil\right).
\end{aligned}$$

So any program $P \in \mathcal{P}_{\mathrm{con}}^{\leq c,t,l,o}$ can be encoded using at most

$$cl(o+1) + 2c\left(l\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right) + \lceil a \log_2 t\rceil\right)$$

bist and therefore the VC–dimension of $\mathcal{P}_{\mathrm{con}}^{\leq c,t,l,o}$ can be estimated as stated in the following theorem.

Theorem 8.2.3

Let c, t, l and o be nonnegative integers. Then

$$\mathrm{VCD{\small IM}}\left(\mathcal{P}_{\mathrm{con}}^{\leq c,t,l,o}\right) = \mathcal{O}\left(cl(o+1) + 2c\log_2\left((p^2+5p)^l t^a\right)\right).$$

Proof. The claim is due to

$$\begin{aligned} & cl(o+1) + 2c\left(l\left(\lceil \log_2(p+5) + \lceil \log_2 p\rceil\right) + \lceil a \log_2 t\rceil\right) \\ = \;& \mathcal{O}\left(cl(o+1) + 2c\left(l \log_2(p +^2 +5p) + a\log_2 t\right)\right) \\ = \;& \mathcal{O}\left(cl(o+1) + 2c\left(\log_2 (p^2+5p)^l + \log_2 t^a\right)\right) \\ = \;& \mathcal{O}\left(cl(o+1) + 2c\log_2\left((p^2+5p)^l t^a\right)\right) \end{aligned}$$

and an application of Lemma 8.1.1. □

The VC–Dimension of range–restricted Prolog(+T)–programs

In some sense this situation is similar to the situation from the last section. Assume that $P = \{P_1, \ldots, P_k\} \in \mathcal{P}_{\mathrm{rr}}^{\leq c,t,l,o}$ is given such that $P_i = \varphi_i$:$-\psi_1^{(i)}, \ldots, \psi_{n_i}(i)$. Similarly as in the case of constrained programs it suffices to encode the terms from $\left\{\psi_1^{(i)}, \ldots, \psi_{n_i}^{(i)}\right\}$. Since $\left|\left\{\psi_1^{(i)}, \ldots, \psi_{n_i}^{(i)}\right\}\right| \leq l-1$ the complete tail of P_i can be encoded using at most

$$(l-1)\left(o+1+2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil + \lceil a\log_2 t\rceil\right)\right)$$

bits while the remaining literal φ_i can be encoded by skipping the involved terms using at most

$$o+1+2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right)$$

bits. So the complete rule P_i can be encoded with

$$o+1+2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right)+(l-1)\left(o+1+2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil + \lceil a\log_2 t\rceil\right)\right)$$

bits. We have

$$\begin{aligned}
& o+1+2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right) \\
& +(l-1)\left(o+1+2\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil + \lceil a\log_2 t\rceil\right)\right) \\
= \ & o+1+2\lceil \log_2(p+5)\rceil + 2\lceil \log_2 p\rceil + (l-1)(o+1) + 2(l-1)\lceil \log_2(p+5)\rceil \\
& +2(l-1)\lceil \log_2 p\rceil + 2(l-1)\lceil a\log_2 t\rceil \\
= \ & l(o+1)+2\left(l\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right) + (l-1)\lceil a\log_2 t\rceil\right).
\end{aligned}$$

Therefore P can be encoded using at most

$$cl(o+1)+2c\left(l\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right) + (l-1)\lceil a\log_2 t\rceil\right)$$

bits and we can estimate the VC–dimension of $\mathcal{P}_{\mathrm{rr}}^{\leq c,t,l,o}$ as follows.

Theorem 8.2.4

Let c, t, l and o be nonnegative integers. Then

$$\mathrm{VCDIM}\left(\mathcal{P}_{\mathrm{rr}}^{\leq c,t,l,o}\right) = \mathcal{O}\left(cl(o+1)+2c\log_2\left((p^2+5p)^l t^{a+l-1}\right)\right).$$

Proof. The claim is due to

$$cl(o+1)+2c\left(l\left(\lceil \log_2(p+5)\rceil + \lceil \log_2 p\rceil\right) + (l-1)\lceil a\log_2 t\rceil\right)$$

$$\begin{aligned}
&= \mathcal{O}\left(cl(o+1) + 2c\left(l\left(\log_2(p+5) + \log_2 p\right) + (l-1)a\log_2 t\right)\right) \\
&= \mathcal{O}\left(cl(o+1) + 2c\left(l\left(\log_2(p^2+5p)\right) + (l-1)a\log_2 t\right)\right) \\
&= \mathcal{O}\left(cl(o+1) + 2c\left(\log_2(p^2+5p)^l + \log_2 t^{a+l-1}\right)\right) \\
&= \mathcal{O}\left(cl(o+1) + 2c\log_2\left((p^2+5p)^l t^{a+l-1}\right)\right)
\end{aligned}$$

and Lemma 8.1.1. □

Having obtained the above results on the complexity of the identification task for PROLOG(+T)–programs our treatment of first order inductive temporal logic programming is complete. We have seen the following:

- It is reasonable to study programs written in PROLOG(+T) since this language is both powerful (it contains the full first order fragment of horn clause programs) and still relatively tractable (due to the syntactic limitation to rules),
- PROLOG(+T) is equipped with a well defined semantics given by temporally closed sets of ground atoms and
- PROLOG(+T) allows specialization and generalization of concepts by application of refinement operators.

All these points make clear that PROLOG(+T) is a suitable language for the specification of reactive systems using first order temporal logic. In contrast a restriction to propositional temporal logic results in the language LTL which is decidable for satisfiability and which is not limited to formulas in clause form. Although LTL is consequently less expressive than PROLOG(+T) the two properties mentioned before justify studying the identification problem in LTL. This will therefore be the topic of the following part of this thesis.

Part III.

Propositional Inductive Temporal Logic Programming

9. Preliminaries

Contents

9.1. Finite Automata on infinite Objects 192

9.2. Automata Constructions for Propositional Temporal Logic Formulas 196

9.2.1. A Modified Formal Automata–Model 196

9.2.2. A primitive Construction 199

9.2.3. An Overview over improved Constructions 203

9.2.4. Some Complexity Results 205

9.2.5. Checking Language–Emptiness 210

Having completed our treatment of first order inductive temporal logic programming we will now restrict ourselves on propositional logic languages. On the one hand this will be a limitation since we do not allow reasoning about functions and predicates with arity greater than 0 but on the other hand we will be able to use much more syntactically complex statements since we will not be limited to statements in clause form any longer. The language of interest will be LTL as introduced in chapter 2.3.1.

In this chapter we will describe the necessary preliminaries from the theory of propositional (linear) temporal logic which will be used in the sequel. First we will introduce the concept of Büchi–automata which are automata over infinite sequences of symbols (*infinite words* or *ω–words*). After this we will see that every LTL–formula φ can be translated into a Büchi automaton which has a nonempty accepted language if and only if the formula φ from which is has been constructed is satisfiable.

9.1. Finite Automata on infinite Objects

We will now introduce notations which allow us to generalize the theory of formal languages consisting of finite words to such languages which consist of infinite words, that is words which can be seen as an infinite sequence of elements (called the *letters* of the word). Therefore assume for the rest of this chapter that Σ is a finite set of symbols, called the *alphabet*.

Definition 9.1.1 (ω–word, e.g. Lothaire [106])
An *infinite word* (or *ω–word*) over Σ is a mapping $w : \mathbb{N} \to \Sigma$. The set of all infinite words over Σ is denoted as Σ^ω. Every set $L \subseteq \Sigma^\omega$ is called an *ω–language* over Σ.

ω–languages are a natural way to extend the theory of formal languages to infinite sequences of letters. In many practical applications ω–languages are used in order to model infinite sequences of actions performed by nonterminating (reactive) systems. The set of *all* possible behaviors of such a system is described in terms of an ω–language and it is then checked if a property φ holds in this system simply by checking if the ω–language which is recognized by the product structure which emerges from the model of the system and the negation of the formula, is empty. How to achieve this, will the subject of the rest of this chapter.

In order to recognize ω–languages, the theory of finite automata has been extended by adding structures which allow acceptance of infinite words. This leads to the theory of ω–automata as described by Büchi (see [26]), Street (see [155]), Muller (see [123]) and Rabin (see [138]). We will describe ω–automata in a similar was as defined by Büchi since this type of automata is used in model checking to characterize the set of models of LTL–formulas.

Definition 9.1.2 (Büchi–automaton, e.g. Wolper [172])

Let Σ be an alphabet. A *nondeterministic Büchi–automaton* over Σ is a tuple

$$\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, S_f)\,,$$

where

- $\textsc{States}$ is a finite set (the set of *states*),
- $\delta : \textsc{States} \times \Sigma \to 2^{\textsc{States}}$ is the *transition relation*,
- $S_0 \subseteq \textsc{States}$ is the set of *initial states* and
- $S_f \subseteq \textsc{States}$ is the set of *accepting* or *final states*.

Note that the formal definition of such a Büchi automaton does only slightly differ from the definition of a finite state automaton accepting finite words. In order to accept infinite words over Σ the acceptance condition of $\mathcal{A}$ has to be modified since it is clear that the acceptance condition for finite words, requiring that the consuming the last symbol leads into a final state, cannot be applied (since there is no *last symbol*). So assume that a Büchi automaton $\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, S_f)$, an ω–word w and a mapping $\gamma : \mathbb{N} \to \textsc{States}$ are given. We will denote the set of all states occurring infinitely often in the sequence labeled by γ as $S^\infty(\gamma)$. Formally:

$$S^\infty(\gamma) = \{s \in \textsc{States} \mid \|\{i \mid \gamma(i) = s\}\| = \infty\}\,.$$

We call γ *adequate for* $(\mathcal{A}, w)$ (or $(\mathcal{A}, w)$*–adequate*) if and only if γ satisfies the following properties:

1. $\gamma(0) \in S_0$,

2. for every $i \geq 0$ it holds that $\gamma(i+1) \in \delta(\gamma(i), w(i))$ and

3. $S^\infty(\gamma) \cap S_f \neq \emptyset$.

A word $w \in \Sigma^\omega$ is said to be *accepted by* $\mathcal{A}$ if and only if there is a mapping γ such that γ is adequate for $(\mathcal{A}, w)$. Similarly a subset L of Σ^ω (that is, an ω–language) is accepted by $\mathcal{A}$ if and only if there is a mapping γ which is such that γ is $(\mathcal{A}, w)$–adequate for *every* $w \in L$.

It has proven useful to define a slight modification of Büchi–automata in which the last component of the tuple is not a set of states but a set of sets of states. This leads to *generalized Büchi–automata.* If

$$\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$$

is such a generalized Büchi–automaton and $w \in \Sigma^\omega$ is an infinite word over Σ, then $\gamma : \mathbb{N} \to \textsc{States}$ is called *adequate for* $(\mathcal{A}, w)$ (or $(\mathcal{A}, w)$–adequate) if any only if

1. $\gamma(0) \in S_0$,

2. for every $i \geq 0$ it holds that $\gamma(i+1) \in \delta(\gamma(i), w(i))$ and

3. if $\mathcal{F} = \{F_1, \ldots, F_n\}$, then $S^\infty(\gamma) \cap F_i \neq \emptyset$ for $i = 1, \ldots, n$.

Again we will call w accepted by $\mathcal{A}$ if there is γ such that γ is $(\mathcal{A}, w)$–adequate and $L \subseteq \Sigma^\omega$ is called *accepted by* $\mathcal{A}$ if there is γ such that γ is $(\mathcal{A}, w)$–adequate for every $w \in L$.

In both cases, that is if $\mathcal{A}$ is a Büchi–automaton or if $\mathcal{A}$ is a generalized Büchi–automaton, the language $L(\mathcal{A})$ is defined as the set of all ω–words which are accepted by $\mathcal{A}$. $L \subseteq \Sigma^\omega$ is called *(generalized) Büchi–acceptable* if and only if there is a (generalized) Büchi–automaton $\mathcal{A}$ such that $L = L(\mathcal{A})$. From [172] we have the following lemma.

Lemma 9.1.1 (e.g. Wolper [172])

Let $L \subseteq \Sigma^\omega$ be any ω–language. Then L is Büchi–acceptable if and only if L is generalized Büchi–acceptable.

Proof. The only–if part is immediately: if $\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, S_f)$ is a Büchi–automaton, then we define $\bar{\mathcal{A}} = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ with $\mathcal{F} = \{\{s_1, \ldots, s_n\}\}$ for $S_f = \{s_1, \ldots, s_n\}$. Then $\bar{\mathcal{A}}$ is a generalized Büchi–automaton and it is straightforward to show that $w \in L(\mathcal{A})$ if and only if $w \in L(\bar{\mathcal{A}})$ for every $w \in \Sigma^\omega$. For the if–part assume that $\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ with $\mathcal{F} = \{F_1, \ldots, F_n\} \subseteq 2^{\textsc{States}}$ is given. We then define $\bar{\mathcal{A}} = \big(\Sigma, \textsc{States}', \delta', S_0', S_f\big)$ as follows:

- $\textsc{States}' = \{(s, i) \mid s \in \textsc{States}, i = 1, \ldots, n\}$,
- $S_0' = \{(s, 1) \mid s \in S_0\}$,
- for all $s, t \in \textsc{States}$, for all $i, j \in \{1, \ldots, j\}$ and each $\sigma \in \Sigma$ define $(t, i) \in \delta'((s, j), a)$ if and only if $t \in \delta(s, a)$ and $i = j$ if $s \notin F_j$ respectively $i = j + 1 \mod k$ if $s \in F_j$ and
- $S_f = \{(s, 1) \mid s \in F_1\}$.

It is now straightforward to prove that $w \in L(\mathcal{A})$ if and only if $w \in L(\bar{\mathcal{A}})$ for every $w \in \Sigma^\omega$. So the lemma is proved. □

Since Büchi–acceptance and generalized Büchi–acceptance are equivalent, it is sufficient to concentrate on algorithms which construct generalized Büchi–automata and apply the construction from the proof of Lemma 9.1.1 to the resulting automaton.

The usefulness of nondeterministic Büchi–automata also comes from the fact that they are closed under every boolean operation. We have the following properties.

Theorem 9.1.1 (e.g. Clarke et al. [35] and Sistla et al. [149])

1. If $L_1 \subseteq \Sigma^\omega$ and $L_2 \subseteq \Sigma^\omega$ are Büchi–acceptable languages, then so are

   a) $L_1 \cap L_2$ and

b) $L_1 \cup L_2$

and

2. if $L \subseteq \Sigma^\omega$ is Büchi–acceptable, then so is $\bar{L} = \Sigma^\omega \setminus L$.

We will not prove this theorem but we refer to section 9.2.2 for a construction of the *product* of two Büchi–automata which results in an automaton which accepts the intersection of the languages accepted by the original automata.

9.2. Automata Constructions for Propositional Temporal Logic Formulas

The reason for studying Büchi–automata stems from the fact that from each LTL–formula one can construct an automaton which accepts exactly the sequences of states which are models of this formula. So we will now study a language which differs from the language PROLOG(+T) studied in the last part in two ways:

1. it is a propositional logic based temporal logic language and
2. it is not limited to sets of clauses which contain at least one positive literal.

In fact we will study the full language LTL from now in. Therefore assume that a finite set X of *proposition symbols* is given. We will refer to elements of X by writing $p, q, \ldots$ sometimes using indexes. *Formulas* of the language LTL are defined as in chapter 2.3. Note that in contrast to FOLTL we do not deal with the operator P here but instead we use the operator R.

9.2.1. A Modified Formal Automata–Model

In this section we will describe a slight modification of the concept of Büchi–automata which is needed in order to allow the manipulation of such automata during the process of formula refinement as described later. In *classical* construction procedures for automata from LTL–formulas (see e.g. [172]) states are labeled with formulas from a set of subformulas of the original formula φ. This set of subformulas is usually referred to as the *closure* of φ. The problem which we face here is that certain formulas might appear more than once (and therefore at different positions) in φ. Consider for example the formula $\varphi = \mathsf{GF}\left((\mathsf{F}p \rightarrow q) \vee (r \rightarrow \mathsf{F}p)\right)$. As a subformula, the formula $\psi = \mathsf{F}p$ is treated as a single element. But for refining the formula φ it can make a difference if the refined formula is for example $\varphi_1 = \mathsf{GF}\left((\mathsf{X}p \rightarrow q) \vee (r \rightarrow \mathsf{F}p)\right)$ or $\varphi_2 = \mathsf{GF}\left((\mathsf{F}p \rightarrow q) \vee (r \rightarrow \mathsf{X}p)\right)$. So we need some model which allows to store additional information regarding the positions at which certain formulas occur.

For the construction we will assume that every LTL–formula is in *negation–normal–form*, that is it contains only the operators X, U and R, the connectives $\wedge$ and $\vee$ together with the constant symbols `true` and `false` and negations only occur in front of propositional symbols. In order to obtain the negation–normal–form $\mathrm{NNF}(\varphi)$ of a formula φ we will have to exploit semantical identities (de–Morgan's laws) and properties of the temporal operators. In particular we will need the following equivalences:

$$\begin{aligned} \mathsf{F}\varphi &\equiv \texttt{true}\mathsf{U}\varphi \text{ and} \\ \mathsf{G}\varphi &\equiv \texttt{false}\mathsf{R}\varphi. \end{aligned}$$

and

$$\begin{aligned} \neg\left(\varphi_1 \mathsf{U} \varphi_2\right) &\equiv \left(\varphi_1\right) \mathsf{R} \left(\varphi_2\right) \text{ and} \\ \neg\left(\varphi_1 \mathsf{R} \varphi_2\right) &\equiv \left(\varphi_1\right) \mathsf{U} \left(\varphi_2\right). \end{aligned}$$

Example 9.2.1

For $\varphi = \neg(p \to \mathsf{X}q)\mathsf{U}\mathsf{F}r$ we have

$$\begin{aligned}
\mathrm{NNF}(\varphi) &= \mathrm{NNF}(\neg(p \to \mathsf{X}q)\mathsf{U}\mathsf{F}r) \\
&= \mathrm{NNF}(\neg(p \to \mathsf{X}q))\mathsf{U}\mathrm{NNF}(\mathsf{F}r) \\
&= (\mathrm{NNF}(p) \wedge \mathrm{NNF}(\neg\mathsf{X}q))\,\mathsf{U}\mathrm{NNF}\,(\mathtt{true}\mathsf{U}r) \\
&= (p \wedge \mathsf{X}\mathrm{NNF}(\neg q))\,\mathsf{U}\,(\mathrm{NNF}(\mathtt{true})\mathsf{U}\mathrm{NNF}(r)) \\
&= (p \wedge \mathsf{X}\neg q)\,\mathsf{U}\,(\mathtt{true}\mathsf{U}r)\,.
\end{aligned}$$

So assume without loss of generality that φ is in negation–normal–form. The *closure* of φ is defined to be the set $\textsc{Closure}(\varphi)$ of all pairs (ψ, p) where ψ is an $\textsc{Ltl}$–formula and $p \in \mathbb{N}^*$ such that $\textsc{Closure}(\varphi)$ satisfies the following conditions:

- $(\varphi, \varepsilon) \in \textsc{Closure}(\varphi)$,
- if $(\varphi_1 \wedge \cdots \wedge \varphi_n, p) \in \textsc{Closure}(\varphi)$, then $(\varphi_1, p1), \ldots, (\varphi_n, pn) \in \textsc{Closure}(\varphi)$,
- if $(\varphi_1 \vee \cdots \vee \varphi_n, p) \in \textsc{Closure}(\varphi)$, then $(\varphi_1, p1), \ldots, (\varphi_n, pn) \in \textsc{Closure}(\varphi)$,
- if $(\mathsf{X}\psi, p) \in \textsc{Closure}(\varphi)$, then $(\psi, p1) \in \textsc{Closure}(\varphi)$,
- if $(\varphi_1\mathsf{U}\varphi_2, p) \in \textsc{Closure}(\varphi)$, then $(\varphi_1, p1), (\varphi_2, p2) \in \textsc{Closure}(\varphi)$ and
- if $(\varphi_1\mathsf{R}\varphi_2, p) \in \textsc{Closure}(\varphi)$, then $(\varphi_1, p1), (\varphi_2, p2) \in \textsc{Closure}(\varphi)$.

So $\textsc{Closure}(\varphi)$ contains all the (not necessarily proper) subformulas of φ together with their positions on φ.

The *states* of the generalized Büchi–automaton $\mathcal{A}_\varphi$ are now defined as certain subsets of $\textsc{Closure}(\varphi)$ which satisfy several semantical constraints.

9.2.2. A primitive Construction

We will now see how the automaton $\mathcal{A}_\varphi$ can be constructed from φ if φ is in negation–normal–form.

The *alphabet* of the automaton will be the set of subsets of symbols occurring in the original formula. The automaton will be a generalized Büchi–automaton, so it will have a set of sets of accepting states. Assume that φ is a formula which contains exactly the symbols of some set X. Then the alphabet of the automaton $\mathcal{A}_\varphi$ is $\Sigma = 2^X$.

The set $\textsc{States}$ of states is now given as the elements s from $\textsc{Seq}(\textsc{Closure}(\varphi)) \times 2^{\mathbb{N}^*}$[1] such that $s = (\Phi, \textsc{Pos}) = \left(\underbrace{\{\varphi_1, \ldots, \varphi_n\}}_{=\Phi}, \underbrace{\{p_1, \ldots, p_m\}}_{=\textsc{Pos}} \right)$ has the following properties:

1. $\texttt{false} \notin \Phi$,
2. $n = m$,
3. for each i: $(\varphi_i, p_i) \in \textsc{Closure}(\varphi)$,
4. for each i: if $\varphi_i = \varphi_1^{(i)} \wedge \varphi_2^{(i)}$, then $\left(\varphi_1^{(i)}, p_i 1\right) \in s$ and $\left(\varphi_2^{(i)}, p_i 2\right) \in s$,
5. for each i: if $\varphi_i = \varphi_1^{(i)} \vee \varphi_2^{(i)}$, then $\left(\varphi_1^{(i)}, p_i 1\right) \in s$ or $\left(\varphi_2^{(i)}, p_i 2\right) \in s$ and
6. for each i: if $\varphi \in X \cup \{\texttt{true}\}$, then $\{p \in \textsc{Pos} \mid \varphi|_p = \varphi_i\} = \textsc{Occ}(\varphi_i, \varphi)$.

Here $(\varphi, p) \in s$ denotes the fact that there is $j \in \{1, \ldots, n\}$ such that $\varphi_j = \varphi$ and $p_j = p$.

What remains to be defined are the transition relation δ, the set S_0 and the set $\mathcal{F}$ of accepting sets of states. δ has the form

$$\delta : \textsc{States} \times 2^P \to 2^{\textsc{States}}.$$

Now let $s_1, s_2 \in \textsc{States}$ be given such that $s_1 = (\Phi, \textsc{Pos}) = (\{\varphi_1, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$

[1]Here $\textsc{Seq}(\textsc{Closure}(\varphi))$ denotes the set of all sequences of elements from $\textsc{Closure}(\varphi)$.

and $s_2 = (\Phi', \textsc{Pos}') = (\{\varphi'_1, \dots, \varphi'_m\}, \{p'_1, \dots, p'_m\})$. Furthermore let $\sigma \in \Sigma$ be fixed. Then $s_2 \in \delta(s_1, \sigma)$ iff

1. for each $p \in X$:

   a) $p \in \Phi$ implies $p \in \sigma$ and

   b) $\neg p \in \Phi$ implies $p \notin \sigma$,

2. for each $\psi = \mathsf{X}\bar{\varphi} \in \Phi$ and each $p_i \in \textsc{Pos}$ such that $\varphi_i = \psi$ there is $j_i \in \{1, \dots, m\}$ such that $\varphi'_{j_i} = \bar{\varphi}$ and $p'_{j_i} = p_i 1$,

3. for each $\psi = \bar{\varphi}_1 \mathsf{U} \bar{\varphi}_2 \in \Phi$ and each $p_i \in \textsc{Pos}$ such that $\varphi_i = \psi$ there is either $j_i \in \{1, \dots, n\}$ such that $\varphi'_{j_i} = \bar{\varphi}_2$ or there are $j_{i_1} \in \{1, \dots, n\}, j_{i_2} \in \{1, \dots, m\}$ such that $\varphi_{j_{i_1}} = \bar{\varphi}_1$ and $\varphi'_{j_{i_2}} = \mathsf{X}\psi = \mathsf{X}\bar{\varphi}_1 \mathsf{U} \bar{\varphi}_2$ and

4. for each $\psi = \bar{\varphi}_1 \mathsf{R} \bar{\varphi}_2 \in \Phi$ and each $p_i \in \textsc{Pos}$ such that $\varphi_i = \psi$ there is $j_{i_1} \in \{1, \dots, n\}$ such that $\varphi_{j_{i_1}} = \bar{\varphi}_2$ and there is either $j_{i_2} \in \{1, \dots, n\}$ such that $\varphi_{j_{i_2}} = \bar{\varphi}_1$ or there is $j_{i_2} \in \{1, \dots, m\}$ such that $\varphi'_{j_{i_2}} = \mathsf{X}\psi = \mathsf{X}\bar{\varphi}_1 \mathsf{R} \bar{\varphi}_2$.

The set S_0 is defined as the set of all states such that the original formula φ is contained in its Φ–component, that is

$$S_0 = \{s = (\Phi, \textsc{Pos}) \in \textsc{States} \mid \varphi \in \Phi\}.$$

To define the set $\mathcal{F}$ of sets of accepting states we define the concept of *eventualities*. Eventualities are formulas which are needed in order to guarantee that given a formula $\varphi_1 \mathsf{U} \varphi_2$, the formula φ_2 is indeed fulfilled at some point of time. So if $(\varphi_1 \mathsf{U} \varphi_2, p) \in \textsc{Closure}(\varphi)$, then $e(\varphi_2) = \varphi_1 \mathsf{U} \varphi_2$ is called an *eventuality*. Now assume that $\textsc{Closure}(\varphi)$ contains exactly the eventualities $(e_1(\varphi_1), p_1), \dots, (e_k(\varphi_k), p_k)$. Then we define $\mathcal{F} = \{F_1, \dots, F_k\}$ with

$$F_i = \{s = (\Phi, \textsc{Pos}) \in \textsc{States} \mid \{e_i(\varphi_i), \varphi_i\} \subseteq \Phi \text{ or } e_i(\varphi_i) \notin \Phi\}.$$

The construction of $\mathcal{A}_\varphi$ is now complete. This construction can be extended from single formulas to sets of formulas in the obvious way. Let $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ be any set of LTL–formulas. Then

$$\mathcal{A}_\Phi = \mathcal{A}_{\varphi_1} \times \cdots \times \mathcal{A}_{\varphi_n}.$$

where $\times$ is a (slightly more complicated) generalization of the *product construction* from finite automata to Büchi–automata.

This construction can be carried out as follows: first we construct the set STATES from the sets of states of the input automata and after this we extract the remaining components from this set. How to achieve this is described in section 9.2.4.

Now let $\mathcal{A}_{\varphi_1} = (\Sigma, \text{STATES}_1, \delta_1, S_{0,1}, \mathcal{F}_1)$ and $\mathcal{A}_{\varphi_2} = (\Sigma, \text{STATES}_2, \delta_2, S_{0,2}, \mathcal{F}_2)$ be Büchi–automata representing LTL–formulas φ_1 and φ_2. In order to respect the positions of the original formulas in the *new* formula $\varphi_1 \wedge \varphi_2$ we have to change the positions in the original states from STATES_1 and STATES_2. Therefore we replace STATES_1 and STATES_2 by

$$\bigcup_{s \in \text{STATES}_1} \left(\Phi(s), \bigcup_{p \in \text{POS}(s)} 1p \right)$$

and

$$\bigcup_{s \in \text{STATES}_2} \left(\Phi(s), \bigcup_{p \in \text{POS}(s)} 2p \right).$$

Furthermore assume that

$$\begin{aligned} \text{STATES}_1 &= \left\{ s_1^{(1)}, \ldots, s_{n_1}^{(1)} \right\} \text{ and} \\ \text{STATES}_2 &= \left\{ s_1^{(2)}, \ldots, s_{n_2}^{(2)} \right\} \end{aligned}$$

and

$$S_{0,1} = \left\{ s_{i_1}^{(1)}, \ldots, s_{i_k}^{(1)} \right\} \text{ and}$$

$$S_{0,2} = \left\{ s^{(2)}_{j_1}, \dots, s^{(2)}_{2_k} \right\}$$

Then $\mathcal{A} = \mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2}$ is the Büchi–automaton $\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ which has a set States of states constructed as follows:

$$\begin{aligned}
S_0 &= \bigcup_{k_1=1}^{k} \bigcup_{k_2=1}^{l} \left\{ \begin{array}{c} \left(\Phi\left(s^{(1)}_{i_{k_1}}\right) \cup \Phi\left(s^{(2)}_{j_{k_2}}\right) \cup \{\varphi_1 \wedge \varphi_2\}, \right. \\ \left. \textsc{Pos}\left(s^{(1)}_{i_{k_1}}\right) \cup \textsc{Pos}\left(s^{(2)}_{j_{k_2}}\right) \cup \{\varepsilon\} \right) \end{array} \right\}, \\
S &= \bigcup_{i=1}^{n_1} \bigcup_{j=1}^{n_2} \left\{ \left(\Phi\left(s^{(1)}_i\right) \cup \Phi\left(s^{(2)}_j\right), \textsc{Pos}\left(s^{(1)}_i\right) \cup \textsc{Pos}\left(s^{(2)}_j\right) \right) \right\} \text{ and} \\
\textsc{States} &= S_0 \cup S,
\end{aligned}$$

while the remaining components of the automaton (i.e. the transition relation δ and the acceptance component $\mathcal{F}$) have to be extracted from States (see the algorithms in section 9.2.4 for details).

This construction yields an automaton which accepts the language $L(\mathcal{A}_{\varphi_1}) \cap L(\mathcal{A}_{\varphi_2})$. Furthermore we have

$$\begin{aligned}
|\textsc{States}| &= |\textsc{States}_1| \cdot |\textsc{States}_2| + |S_{0,1}| \cdot |S_{0,2}| \text{ and} \\
|S_0| &= |S_{0,1}| \cdot |S_{0,2}|.
\end{aligned}$$

In order to prove that for a set Φ of Ltl–formulas and an Ltl–formula φ the relation $\Phi \models \varphi$ holds, we proceed as follows:

1. For $\Phi = \{\varphi_1, \dots, \varphi_n\}$ we construct $\mathcal{A}_\Phi = \mathcal{A}_{\varphi_1} \times \cdots \times \mathcal{A}_{\varphi_n}$[2],
2. construct $\mathcal{A}_{\neg\varphi}$,
3. construct $\mathcal{A} = \mathcal{A}_\Phi \times \mathcal{A}_{\neg\varphi}$ and

[2]Note that it is also possible to construct $\mathcal{A}_\Phi = \mathcal{A}_{\bigwedge_{i=1}^{n} \varphi_i}$ directly (i.e. without using the product construction).

4. check if $L(\mathcal{A}) = \emptyset$.

For some of the algorithms which we will introduce in a later chapter we will also need a method for constructing the *union* of two Büchi–automata. Again assume that φ_1 and φ_2 are given and assume that $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$ are as above. Then the Büchi–automaton $\mathcal{A}_{\varphi_1}||\mathcal{A}_{\varphi_2}$ representing $\varphi_1 \vee \varphi_2$[3] can be constructed by first modifying $\textsc{States}_1$ and $\textsc{States}_2$ as described above and then constructing the set $\textsc{States}$ by

$$S_0 = \bigcup_{k_1=1}^{k} \bigcup_{k_2=1}^{l} \left\{ \left(\Phi\left(s^{(1)}_{i_{k_1}}\right) \cup \Phi\left(s^{(2)}_{j_{k_2}}\right) \cup \{\varphi_1 \wedge \varphi_2\}, \textsc{Pos}\left(s^{(1)}_{i_{k_1}}\right) \cup \textsc{Pos}\left(s^{(2)}_{j_{k_2}}\right) \cup \{\varepsilon\} \right) \right\}$$

and

$$\textsc{States} = S_0 \cup \textsc{States}_1 \cup \textsc{States}_2.$$

The automaton $\mathcal{A}_{\varphi_1}||\mathcal{A}_{\varphi_2}$ is then given as $(\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ (again with δ and $\mathcal{F}$ extracted from $\textsc{States}$). It is easily seen that this construction is sound. Furthermore we have

$$|\textsc{States}| = |S_{0,1}| \cdot |S_{0,2}| + |\textsc{States}_1| + |\textsc{States}_2|$$

9.2.3. An Overview over improved Constructions

The primitive construction presented in the last section always yields an automaton whose state set is of size exponential in the length of the input formula. We will therefore give an overview over several optimization techniques which allow the construction of *smaller* automata.

Removing Transitions A transition which is not explicitly needed (and which is therefore *redundant*) in the automaton can be deleted. Such transitions can be identified as follows: if there is $s \in \textsc{States}$, $\sigma \in \Sigma$, $s_1 = (\Phi_1, \textsc{Pos}_1) \in \textsc{States}$ and $s_2 = (\Phi_2, \textsc{Pos}_2) \in \textsc{States}$ such that

[3]That is: $\mathcal{A}_{\varphi_1}||\mathcal{A}_{\varphi_2} = \mathcal{A}_{\varphi_1 \vee \varphi_2}$.

1. $s_1 \in \delta(s, \sigma)$, $s_2 \in \delta(s, \sigma)$ and
2. for every $\varphi_1 \mathsf{U} \varphi_2 \in \Phi_1$: $\varphi_1 \mathsf{U} \varphi_2 \in \Phi_2$ and $\varphi_2 \in \Phi_1$ implies $\varphi_2 \in \Phi_2$,

then the transition from s to s_2 can be deleted from the automaton (see [172] for a justification of this optimization).

Eliminating equivalent states By identifying sets of formulas which are in some sense *equivalent* it is possible to lower the number of states. Consider for example the state s given by $s = (\{\varphi_1, \varphi_2, \varphi_2 \wedge \varphi_2\}, \textsc{Pos})$. This state has the property that $\Phi \models \varphi$ iff $\Phi' = \Phi \setminus \{\varphi_1 \wedge \varphi_2\} = \{\varphi_1, \varphi_2\} \models \varphi$ for every Ltl–formula φ. This allows us to remove the original state s and replace it with some state $s' = (\Phi', \textsc{Pos}')$. In [76], [172] and several other papers the following improvements have been discussed:

1. $\{\varphi_1, \varphi_2, \varphi_2 \wedge \varphi_2\} \rightarrow \{\varphi_1, \varphi_2\}$,
2. $\{\varphi_1, \varphi_1 \vee \varphi_2\} \rightarrow \{\varphi_1\}$,
3. $\{\varphi_2, \varphi_1 \vee \varphi_2\} \rightarrow \{\varphi_2\}$ and
4. $\{\varphi_2, \varphi_1 \mathsf{U} \varphi_2\} \rightarrow \{\varphi_1\}$.

Several other simplification techniques may be applicable. We do not give a more in depth–treatment here since we do not need all these techniques in the sequel.

A number of approaches has been introduced for the construction of Büchi–automata from Ltl–formulas. Probably the most straightforward and simple construction (which is the basis for our automaton model) can for example be seen in [172] although its origin comes from [168] and [167] continuing the work originally started by Büchi in [26]. The early constructions of Büchi–automata for Ltl–formulas were of exponential size in the size of the original formula. More sophisticated constructions have been developed in [76], [39], [150] and [72]. [17] and [60] introduce similar approaches for the problem of Ltl model checking which do not construct Büchi–automata directly but which use similar techniques and results.

Besides constructing automata several other approaches for checking LTL–formulas for satisfiability resp. unsatisfiability have been presented. An early paper by Venkatesh (see [169]) describes the construction of a normal form for LTL–formulas which is then given to a resolution–style theorem proving procedure as an input. Similarly Fisher (see [64]) introduces another normal form called *separated normal form* for formulas including future and past operators. Again the theorem proving procedure is based on resolution. Another treatment of this procedure can be found in a paper by Dixon, Fisher and Peim (see [65]). Dixon (see [48] and [47]) and Dixon and Fisher (see [49]) also addressed the topic of speeding up theorem proving procedures in order to improve the satisfiability tests.

Another approach for LTL is presented by Felty (see [61]). Here the calculus for checking formulas is based on the sequent calculus originally introduced by Gentzen (see e.g. [75] or [89]). A further paper extends this sequent calculus from LTL to the modal logic S4.3 (see [62]).

The third popular approach is based on tableaux–style techniques similar to our techniques from chapter 5.3. This technique has been developed by Manna and Wolper (see [171] and [111]) as well as by Lichtenstein, Pnueli and Zuck (see [102] and [103]). Good surveys of tableaux techniques have been presented by Emerson (see [57]) and Reynolds and Dixon (see [139]).

All these approaches have their own powers and weaknesses. But for our purposes the automata–based approach seems to be the most promising one as it allows the generalization and specialization of given formulas from their representing automata as we will see in chapter 10.

9.2.4. Some Complexity Results

In order to estimate the complexity of the refinement procedures to be introduced in the following chapter we will present some results regarding the complexity of some *basic* operations on Büchi–automata. Assume that $\textsc{States}$ is a set of sates which has been constructed. We will sketch the complexity of extracting δ, S_0 and $\mathcal{F}$ from $\textsc{States}$.

Extracting the Transition Relation

Obviously the complexity of the extraction of δ from $\textsc{States}$ has to depend on the number of states, i.e. $|\textsc{States}|$ and the number of elements in the alphabet of the automaton, i.e. $|\Sigma| = |2^X| = 2^{|X|}$. Algorithm 11 is a straightforward implementation of the definition of the transition relation for Büchi–automata. For notational simplicity we will use the following abbreviation. If $s = (\Phi, \textsc{Pos})$ is a state, then $\Phi(s)$ will denote the set of formulas stored in s, i.e. $\Phi(s) = \Phi$. Furthermore assume that $\delta(s, \sigma) = \emptyset$ holds as an initial condition.

We will now give a detailed analysis of the runtime of Algorithm 11. Therefore assume that $n_{\max}$ denotes the maximum number of formulas stored in any element of $\textsc{States}$, that is $n_{\max} = \max\{|\Phi| \mid \Phi = \Phi(s), s \in \textsc{States}\}$. Furthermore we will assume that each check of the form $\varphi \in \Phi(s)$ is atomic, i.e. is can be performed in *one* computation step and checks performed in conditions are performed one–by–one, that is a check which involves n subchecks requires n computation steps.

The part of Algorithm 11 between line 4 and line 11 can then be preformed in

$$\begin{aligned} T_1(|\textsc{States}|, |\Sigma|) &\leq 6|P| \\ &= 6\log_2|\Sigma| \end{aligned}$$

computation steps.

Similarly we have that the part between lines 12 and 32 can be performed in

$$\begin{aligned} T_2(|\textsc{States}|, |\Sigma|) &\leq n_{\max} \cdot (1 + 3n_{\max} + 3 + 4) \\ &= n_{\max} \cdot (8 + 3n_{\max}) \end{aligned}$$

Algorithm 11 Extraction of δ from Σ and States

Input: set States of states and alphabet $\Sigma = 2^X$
Output: transition relation δ

```
1: for each $s_1 \in$ States do
2:   for each $s_2 \in$ States do
3:     for each $\sigma \in \Sigma$ do
4:       for each $p \in P$ do
5:         if $p \in \sigma$ and $p \in \Phi(s_1)$ then
6:           $\delta(s_1, \sigma) \leftarrow \delta(s_1, \sigma) \cup \{s_2\}$
7:         end if
8:         if $p \notin \sigma$ and $\neg p \in \Phi(s_1)$ then
9:           $\delta(s_1, \sigma) \leftarrow \delta(s_1, \sigma) \cup \{s_2\}$
10:        end if
11:      end for
Require:    $s_1 = (\{\varphi_1, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$
Require:    $s_2 = (\{\varphi'_1, \ldots, \varphi'_m\}, \{p'_1, \ldots, p'_m\})$
12:      for $i = 1, \ldots, n$ do
13:        if $\varphi_i = \mathsf{X}\psi$ then
14:          for $j = 1, \ldots, m$ do
15:            if $\varphi'_j = \psi$ and $p'_j = p_i 1$ then
16:              $\delta(s_1, \sigma) \leftarrow \delta(s_1, \sigma) \cup \{s_2\}$
17:            end if
18:          end for
19:        end if
20:        if $\varphi_i = \bar{\varphi}_1 \mathsf{U} \bar{\varphi}_2$ then
21:          if $\bar{\varphi}_2 \in \Phi(s_1)$ or $\bar{\varphi}_1 \in \Phi(s_1)$ and $\mathsf{X}\psi \in \Phi(s_2)$ then
22:            $\delta(s_1, \sigma) \leftarrow \delta(s_1, \sigma) \cup \{s_2\}$
23:          end if
24:        end if
25:        if $\varphi_i = \bar{\varphi}_1 \mathsf{R} \bar{\varphi}_2$ then
26:          if $\bar{\varphi}_2 \in \Phi(s_1)$ then
27:            if $\bar{\varphi}_1 \in \Phi(s_1)$ or $\mathsf{X}\psi \in \Phi(s_2)$ then
28:              $\delta(s_1, \sigma) \leftarrow \delta(s_1, \sigma) \cup \{s_2\}$
29:            end if
30:          end if
31:        end if
32:      end for
33:    end for
34:  end for
35: end for
36: return $\delta$
```

$$= \quad 8n_{\max} + 3n_{\max}^2$$

steps.

The complexity of the complete algorithm can therefore be estimated as follows:

$$\begin{aligned} T(|\textsc{States}|, |\Sigma|) &\leq |\textsc{States}|^2 \cdot |\Sigma| \cdot (T_1(|\textsc{States}|, |\Sigma|) + T_2(|\textsc{States}|, |\Sigma|)) \\ &= |\textsc{States}|^2 \cdot |\Sigma| \cdot \left(6 \log_2 |\Sigma| + 8n_{\max} + 3n_{\max}^2\right) \\ &\in \mathcal{O}\left(n_{\max}^2 \cdot |\textsc{States}|^2 \cdot |\Sigma| \cdot \log_2 |\Sigma|\right). \end{aligned}$$

So we have the following theorem.

Theorem 9.2.1

Let $\textsc{States}$ and Σ be given. Then $\delta : \textsc{States} \times \Sigma \to 2^{\textsc{States}}$ can be constructed in time

$$\mathcal{O}\left(n_{\max}^2 \cdot |\textsc{States}|^2 \cdot |\Sigma| \cdot \log_2 |\Sigma|\right).$$

Extracting the Initial States

Extracting the initial states from $\textsc{States}$ is the simplest task. By definition every $s = (\Phi, \textsc{Pos}) \in S_0$ is such that $\varphi \in \Phi$. Then a simple linear search strategy can check if s is indeed contained in S_0. The runtime of such a check is bounded from above by $n_{\max}$. Consequently checking every $s \in \textsc{States}$ can be done in time $\mathcal{O}(n_{\max} \cdot |\textsc{States}|)$.

Theorem 9.2.2

Let $\textsc{States}$ be given. Then $S_0 \subseteq \textsc{States}$ can be extracted in time $\mathcal{O}(n_{\max} \cdot |\textsc{States}|)$.

Extracting the Acceptance Component

For the extraction of the acceptance component $\mathcal{F}$ it is necessary to *collect* the eventualities which are included in the states. This can be accomplished in $n_{\max} \cdot |\textsc{States}|$

steps yielding a set $\textsc{Ev}$ of pairs $(e_i(\varphi_i), p_i)$. The size of $\textsc{Ev}$ is bounded from above by $n_{\max} \cdot |\textsc{States}|$. Following the definition of the acceptance component we can compute $\mathcal{F}$ using Algorithm 12.

Algorithm 12 Extracting the Acceptance Component $\mathcal{F}$ from $\textsc{States}$

Input: set $\textsc{States}$ of states
Output: acceptance component $\mathcal{F}$

```
1: $\mathcal{F} \leftarrow \emptyset$
2: compute $\textsc{Ev}$ (as described)
3: for each $(e_i(\varphi_i), p_i) \in \textsc{Ev}$ do
4:    $F \leftarrow \emptyset$
5:    for each $s \in \textsc{States}$ do
6:       if $e_i(\varphi_i) \in \Phi(s)$ and $\varphi \in \Phi(s)$ or $e_i(\varphi_i) \notin \Phi(s)$ then
7:          $F \leftarrow F \cup \{s\}$
8:       end if
9:    end for
10:   $\mathcal{F} \leftarrow \mathcal{F} \cup \{F\}$
11: end for
12: return $\mathcal{F}$
```

The time complexity of Algorithm 12 can be estimated as follows:

$$\begin{aligned}
T(|\textsc{States}|) &\leq 1 + n_{\max} \cdot |\textsc{States}| + |\textsc{Ev}| \cdot (1 + |\textsc{States}| \cdot (2+1) + 1) + 1 \\
&= 2 + n_{\max} \cdot |\textsc{States}| + |\textsc{Ev}| \cdot (2 + 3 \cdot |\textsc{States}|) \\
&\leq 2 + n_{\max} \cdot |\textsc{States}| + n_{\max} \cdot |\textsc{States}| \cdot (2 + 3 \cdot |\textsc{States}|) \\
&= 2 + 3n_{\max} \cdot |\textsc{States}| + 3n_{\max} \cdot |\textsc{States}|^2 \\
&\in \mathcal{O}\left(n_{\max} \cdot |\textsc{States}|^2\right).
\end{aligned}$$

So we have the following theorem.

Theorem 9.2.3

Let $\textsc{States}$ be given. Then the acceptance component $\mathcal{F}$ can be extracted from $\textsc{States}$ in time $\mathcal{O}\left(n_{\max} \cdot |\textsc{States}|^2\right)$.

9.2.5. Checking Language–Emptiness

We have already mentioned that it will be necessary to check the emptiness of the languages accepted by Büchi–automata. In this section we will see how this can be achieved.

Let $G = (V, E)$ be a directed graph and let $V' \subseteq V$ be a nonempty set of vertices from V and let E' be a nonempty set of edges from E. Then the subgraph $G' = (V', E')$ is called *maximal strongly connected* if

- for each pair n_1, n_2 of vertices from V' it holds that n_2 is reachable from n_1 and n_1 is reachable from n_2 and
- V' is maximal wrt. $\subseteq$, that is there is no $v \in V \setminus V'$ such that $(V' \cup \{v\}, E')$ is strongly connected.

The set of all strongly connected subgraphs of G is called the set of *maximal strongly connected components* of G. The maximal strongly connected components of G form a partition of G into disjoint subsets. A maximal strongly connected component (V', E') is called *non–trivial* if either $|V'| > 1$ or $V' = \{v\}$ for some v and $(v, v) \in E'$.

The maximal strongly connected components of a graph can be computed in time $\mathcal{O}(|V|)$ (see [156] and [74]).

The link to our problem is given as follows: Obviously every extended Büchi–automaton $\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ induces a directed Graph $G_{\mathcal{A}} = (\textsc{States}, E)$ where for every pair $s_1, s_2 \in \textsc{States}$ the edge (s_1, s_2) belongs to E if there is an element $\sigma \in \Sigma$ such that $s_2 \in \delta(s_1, \sigma)$. For checking that $L(\mathcal{A}) = \emptyset$ it suffices to compute the maximal strongly connected components of $G_{\mathcal{A}}$ and check if there is a maximal strongly connected component S which is reachable from some initial state $s_0 \in S_0$ such that $S \cap F \neq \emptyset$ for every $F \in \mathcal{F}$. So checking logical implication of some property φ in a system given by an $\textsc{Ltl}$–program P can be reduced to the computation of the maximal strongly connected components of $\mathcal{A}_P \times \mathcal{A}_{\neg\varphi}$ and a simple containment check.

10. Automata Manipulations

Contents

10.1. Implication as an Ordering 212

10.2. Upward Refinement 213

10.2.1. Formulas without Temporal Operators 214

10.2.2. Formulas with Temporal Operators 216

10.3. Downward Refinement 218

10.3.1. Formulas without Temporal Operators 219

10.3.2. Formulas with Temporal Operators 220

10.4. Modifying Automata by Application of Refinement Operations 221

10.4.1. Upward Refinement 221

10.4.2. Downward Refinement 232

10.4.3. Extracting Formulas 237

10.5. The Identification Process 238

This chapter will deal with the procedures which are necessary in order to refine Ltl–programs. As we have seen in the last chapter, we are always able to construct a generalized Büchi–automaton from a temporal logic formula which has a nonempty accepted language if and only if the formula from which the automaton had been constructed is satisfiable. The construction relies on the set of subformulas occurring in the original formula. So a state is labeled with a set of formulas together with their positions.

Now assume that a modified Büchi–automaton $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ constructed from a Ltl–formula φ is given. By definition we have $\varphi \in \Phi(s)$ for every $s \in S_0$. Now assume further that any Ltl–formula e is given (an example). Then this example can be a positive one or a negative one. In either case we can construct the representing automaton $\mathcal{A}_{\neg e}$.

1. If e is a positive example, then a modification of the model under consideration has to be carried out if $L(\mathcal{A}_\varphi \times \mathcal{A}_{\neg e}) \neq \emptyset$ since in this case the positive example e is not implied by the model under consideration.

2. If e is a negative example, then consequently we have to refine the model if $L(\mathcal{A}_\varphi \times \mathcal{A}_{\neg e}) = \emptyset$ since in this case the negative example e is implied.

In either case we will have to modify the states of the representing automaton $\mathcal{A}_\varphi$ in such a way that the modified model is compatible with the new example e.

10.1. Implication as an Ordering

In the case of FoLtl, we have used the concept of subsumption as the basis for generalization and specialization operations. In Ltl this would not be a good choice for two reasons:

1. Ltl is a propositional temporal logic language, so the concept of substitutions would not make any sense since there are no variable symbols to substitute and

2. Prolog(+T)–objects are essentially clauses. The results regarding greatest specializations and least generalizations are only concerned with such clauses. In Ltl the objects are not limited to clauses and so we cannot hope to apply the techniques established in chapter 6.

So we see that subsumption is not applicable here. However, since Ltl is propositional, we know that the logical consequence relation $\models$ is decidable. This enables us to use a finer relation than the subsumption ordering $\succcurlyeq$, namely the *implication ordering.*

Definition 10.1.1

Let φ_1 and φ_2 be LTL formulas. Then the ordering $\succsim$ is defined as $\varphi_1 \succsim \varphi_2$ if and only if $\varphi_1 \models \varphi_2$.

The notations $\precsim$, $\succ$, $\prec$ are defined in the usual way. In the case of $\varphi_1 \succsim \varphi_1$ and $\varphi_2 \succsim \varphi_1$ we will not introduce a new symbol since in this case $\varphi_1 \equiv \varphi_2$.

10.2. Upward Refinement

We will now show how LTL–formulas can be refined upwards, that is we will see how we can construct a formula ψ from a given formula φ such that $\psi \succsim \varphi$ where ψ is in some sense *minimal.* This concept of minimality will be made precise now.

Definition 10.2.1

Let φ be an LTL–formula. An LTL–formula ψ is called a *minimal upward refinement wrt.* $\succsim$ of φ if

1. $\psi \succsim \varphi$ and
2. there is no LTL–formula ψ' such that $\psi \succ \psi' \succ \varphi$.

Our concept of minimal refinements is identical to the concept of *covers* which is a well–known concept from the theory of ILP (see [126] for example). In other words, a formula ψ is a minimal upward refinement wrt. $\succsim$ of a formula φ if and only if ψ is an upward cover of φ (with respect to the ordering $\succsim$). In particular all properties of covers also hold for minimal refinements.

We will now show how upward covers can be constructed. Therefore assume that φ is a fixed LTL–formula in which propositional variables from the set $X = \{p_1, \ldots, p_m\}$

occur. The construction of refinements of φ depends on the question if there are temporal operators involved in φ.

10.2.1. Formulas without Temporal Operators

Assume that φ is a purely propositional logic formula. The idea of how to refine φ is to construct a formula ψ which has *nearly the same models* as φ. To obtain such a formula we need a special kind of formulas, namely so called *maximal minterms*.

Definition 10.2.2

Let X be a set of propositional variable symbols and let φ be a formula in which exactly the variables from $X = \{p_1, \ldots, p_m\}$ occur. Then φ is called a *maximal minterm* if and only if

1. $\varphi = \bigwedge_{i=1}^{|X|} \psi_i$,
2. for every $i \in \{1, \ldots, k\}$ it holds that $\psi_i \in X$ or $\psi_i \in \{\neg p \mid p \in X\}$ and
3. there is no pair i_0, i_1 such that $\psi_{i_0} \equiv \neg\psi_1$.

The set of all maximal minterms containing variables from X will be denoted as $\textsc{MinTerms}(X)$.

Theorem 10.2.1

Let φ be a propositional logic formula containing variables from a finite set X of propositional symbols. If φ is satisfiable, then for every $\chi \in \textsc{MinTerms}(X)$ with $\varphi \not\models \neg\chi$ a minimal upward refinement of φ is given by $\varphi \wedge \neg\chi$.

Proof. Assume that φ and X are given as required. First we observe that $\varphi \wedge \neg\chi \models \varphi$ for *every* $\chi \in \textsc{MinTerms}(X)$, that is $\varphi \wedge \neg\chi \gtrsim \varphi$ holds. Now assume that χ is chosen such that $\varphi \not\models \neg\chi$. Assume that $\varphi \wedge \neg\chi \not\succ \varphi$, that is assume that $\varphi \gtrsim \varphi \wedge \neg\chi$. Then

$MD(\varphi = MD(\varphi \wedge \neg\chi) = MD(\varphi) \cap MD(\neg\chi)$ or equivalently $MD(\neg\chi) \supseteq MD(\varphi)$. But this gives $\varphi \models \neg\chi$ contradicting the assumptions on φ and χ.

It remains to prove that the formula $\varphi \wedge \neg\chi$ is indeed a minimal refinement. Assume that this is *not* the case, that is assume that there is a formula α such that $\varphi \wedge \neg\chi \succ \alpha \succ \varphi$. Then we have

1. $MD(\varphi \wedge \neg\chi) \subset MD(\alpha) \subset MD(\varphi)$ and
2. $|MD(\neg\chi)| = 2^{|X|} - 1$ since $\chi \in \textsc{MinTerms}(X)$ and therefore $|\mathrm{MD}(\chi)| = 1$.

This gives $|MD(\varphi \wedge \neg\chi)| \leq |MD(\alpha)| \leq |MD(\varphi)|$ since X is assumed to be finite. We can distinguish two cases:

Case 1 $\varphi \models \neg\chi$. Then $|MD(\varphi \wedge \neg\chi)| = |MD(\varphi)|$ and therefore $|MD(\alpha)| = |MD(\varphi)|$ which gives $\alpha \equiv \varphi$ and in particular $\alpha \not\succ \varphi$ which is a contradiction.

Case 2 $\varphi \not\models \neg\chi$. Then we have $|MD(\varphi \wedge \neg\chi)| = |MD(\varphi)| - 1 < |MD(\varphi)|$. But since $MD(\alpha) \subset MD(\varphi)$ and $MD(\varphi \wedge \neg\chi) \subset MD(\alpha)$ this gives either $|MD(\alpha)| = |MD(\varphi)| - 1 = |MD(\varphi \wedge \neg\chi)|$ which yields $\alpha \equiv \varphi \wedge \neg\chi$ or $|MD(\alpha)| = |MD(\varphi)|$ which gives $\alpha \equiv \varphi$. In the former case we have $\varphi \wedge \neg\chi \not\succ \alpha$ and in the latter case we have $\alpha \not\succ \varphi$. So both cases yield a contradiction.

Since every case yields a contradiction, such a formula φ cannot exist and the claim is proved. □

For the sake of simplicity we will introduce a special mapping Ψ_u containing all upward refinements of a formula φ, that is

$$\Psi_u(\varphi) = \{\varphi \wedge \neg\chi \mid \chi \in \textsc{MinTerms}(X)\}.$$

Example 10.2.1

Let $X = \{p_1, p_2, p_3\}$ and $\varphi = p_1 \rightarrow (p_2 \rightarrow p_3)$ be given. Then

$$\textsc{MinTerms}(X) = \left\{\begin{matrix} \neg p_1 \wedge \neg p_2 \wedge \neg p_3, \\ \neg p_1 \wedge \neg p_2 \wedge p_3, \\ \neg p_1 \wedge p_2 \wedge \neg p_3, \\ \neg p_1 \wedge p_2 \wedge p_3 \\ p_1 \wedge \neg p_2 \wedge \neg p_3, \\ p_1 \wedge \neg p_2 \wedge p_3, \\ p_1 \wedge p_2 \wedge \neg p_3, \\ p_1 \wedge p_2 \wedge p_3 \end{matrix}\right\}$$

and

$$\begin{aligned} \Psi_u(\varphi) &= \{(p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge \neg\chi \mid \chi \in \textsc{MinTerms}(X)\} \\ &= \left\{\begin{matrix} (p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (p_1 \vee p_2 \vee p_3), (p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (p_1 \vee p_2 \vee \neg p_3) \\ (p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (p_1 \vee \neg p_2 \vee p_3), (p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (p_1 \vee \neg p_2 \vee \neg p_3) \\ (p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (\neg p_1 \vee p_2 \vee p_3), (p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (\neg p_1 \vee p_2 \vee \neg p_3) \\ (p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (\neg p_1 \vee \neg p_2 \vee p_3), (p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (\neg p_1 \vee \neg p_2 \vee \neg p_3) \end{matrix}\right\}. \end{aligned}$$

10.2.2. Formulas with Temporal Operators

In the foregoing section we have described how to refine purely propositional LTL–formulas. There is one problem with this approach: consider the formula $\varphi = p_1 \wedge p_2$. The refinement procedures for propositional formulas construct formulas such as $(p_1 \wedge p_2) \wedge (\neg p_1 \vee \neg p_2)$, $(p_1 \wedge p_2) \wedge (\neg p_1 \vee p_2)$,... but if the formula to be identified is for example $\psi = \mathsf{G}p_1 \wedge \mathsf{G}p_2$, then this formula will never be constructed. In order to overcome this limitation we will have to describe how to *temporally generalize* formulas. For the case of upward refinement, which will be discussed in this section, consider the following formula $\varphi = \mathsf{F}p$ for some propositional variable symbol $p \in X$. Obviously a temporal interpretation $\mathcal{J} = (s_0, s_1, \dots)$ is a model of φ if there is an index j such that $\mathcal{J}^j \models \varphi$ (or equivalently if $\mathcal{J} \models \mathsf{X}^j\varphi$). So we have that $p \models \mathsf{F}p$. More general $\mathsf{X}^j p \models \mathsf{F}p$ for every $j \geq 0$, that is $\mathsf{X}^j p \gtrsim \mathsf{F}p$ for every $j \in \mathbb{N}$. So an upward refinement of *any* formula $\mathsf{F}\psi$ can be constructed as $\mathsf{X}^j\psi$. This approach can be continued: obviously $\mathsf{G}\psi \models \mathsf{X}^j\psi$, that is $\mathsf{G}\psi \gtrsim \mathsf{X}^j\psi \gtrsim \mathsf{F}\psi$. We will now examine how this idea of temporal refinement can be combined with the refinement procedure described for propositional formulas.

Therefore assume that any LTL–formula φ is given. For the formulas φ_1 and φ_2 we will require that $\psi \gtrsim \varphi_1$ respectively $\psi \gtrsim \varphi_2$ for every formula $\psi \in \Gamma_u(\varphi_1)$ respectively $\psi \in \Gamma_u(\varphi_2)$. Then the set $\Gamma_u(\varphi)$ is defined as follows:

$$\begin{array}{rcl}
\Gamma_u(\varphi) & = & \Psi_u(\varphi) \\
& \cup & \left\{\varphi[\mathsf{X}^i\chi]_p \mid p \in \mathrm{Pos}(\varphi), \varphi|_p = \mathsf{F}\chi \text{ for some } i \geq 0\right\} \\
& \cup & \left\{\varphi[\mathsf{G}\chi]_p \mid p \in \mathrm{Pos}(\varphi), \varphi|_p = \mathsf{X}^i\chi \text{ for some } i \geq 0\right\} \\
& \cup & \left\{ \varphi[\bar{\varphi}_1 \star \varphi_2]_p, \varphi[\varphi_1 \star \bar{\varphi}_2]_p \mid \begin{array}{c} p \in \mathrm{Pos}(\varphi), \varphi|_p = \varphi_1 \star \varphi_2, \bar{\varphi}_1 \in \Gamma_u(\varphi_1), \\ \bar{\varphi}_2 \in \Gamma_u(\varphi_2), \star \in \{\mathsf{U}, \mathsf{R}\} \end{array} \right\}.
\end{array}$$

For proving the properties of the formulas from $\Gamma_u(\varphi)$ we will exploit the following observation:

Lemma 10.2.1 (Replacement–Lemma, Upward–Version)

Let φ and ψ be LTL–formulas, let $p \in \mathrm{Pos}(\varphi)$ be a position in φ. If $\psi \gtrsim \varphi|_p$, then $\varphi[\psi]_p \gtrsim \varphi$.

Proof. Assume that $\varphi|_p \gtrsim \psi$, that is $\varphi|_p \models \psi$. Let $\mathcal{J} \in MD(\varphi)$ be a fixed model of φ. We will prove that $\mathcal{J} \models \varphi[\psi]_p$.

Case 1 If $\mathcal{J} \models \varphi|_p$, then $\mathcal{J} \models \psi$ since $\varphi|_p \gtrsim \psi$. But in this case we also have $\mathcal{J} \models \varphi[\psi]_p$ since $\mathcal{J}(\varphi|_p) = \mathcal{J}(\psi)$.

Case 2 If $\mathcal{J} \not\models \varphi|_p$, then we have to distinguish how $\mathcal{J}$ evaluates ψ. If $\mathcal{J} \not\models \psi$, then the claim is obvious (using the same argumentation as in the foregoing case). So assume that $\mathcal{J} \models \psi$. Then we have two subcases:

Case 2.1 If $\varphi|_p$ is positive in φ, then the claim is immediately.

Case 2.2 If $\varphi|_p$ is negative in φ, then we have $\varphi|_p = \neg q$ for some propositional variable symbol q since φ is assumed to be in negation normal form. But the assumption $\varphi|_p \succsim \psi$ yields $\psi \in \{\texttt{true}, \neg q\}$ and so the claim follows.

This proves the lemma. □

It is now straightforward to prove that every formula contained in the set $\Gamma_u(\varphi)$ is a generalization of φ with respect to the ordering $\succsim$.

Theorem 10.2.2

For every LTL–formula φ and every $\psi \in \Gamma_u(\varphi)$ it holds that $\psi \succsim \varphi$.

Proof. If $\psi \in \Psi_u(\varphi)$, then the claim is due to Theorem 10.2.1. Otherwise the replacement–lemma can be applied. □

10.3. Downward Refinement

Dually to the construction of minimal upward refinements we can construct maximal downward refinements of a formula φ. As we might expect, a maximal downward refinement is defined as follows.

Definition 10.3.1

Let φ be an LTL–formula. An LTL–formula ψ is called a *maximal downward refinement wrt.* $\succsim$ of φ if

1. $\varphi \succsim \psi$ and

2. there is no LTL–formula ψ' such that $\varphi \succ \psi' \succ \psi$.

As in the case of upward refinement we will distinguish between formulas with and without temporal operators. We will see that the concepts developed in the foregoing section can again be adapted.

10.3.1. Formulas without Temporal Operators

Recall that a minimal upward refinement of a formula φ which does not contain temporal operators is given as $\varphi \wedge \neg\chi$ for some maximal minterm χ. The philosophy was as follows: since χ is a minterm, χ has exactly one model. So $\neg\chi$ has $2^{|X|}-1$ models where X is the set of variables under consideration. Building the conjunction of φ and a formula which has as many models as possible without being a tautology (i.e. $\neg\chi$) yields a more general formula which is a minimal upward refinement. The dual aspect of *removing a model* is *adding a model*. So for building a maximal upward refinement of φ we will have to construct a disjunction of φ and a formula which has exactly one model, i.e. a minterm.

Theorem 10.3.1

Let φ be a propositional logic formula containing variables from a finite set X of propositional symbols. If φ is satisfiable but no tautology, then for every $\chi \in \textsc{MinTerms}(X)$ with $\chi \not\models \varphi$ a maximal downward refinement of φ is given by $\varphi \vee \chi$.

Proof. Let φ be any formula containing variables from X. First we will again note that for *every* $\chi \in \textsc{MinTerms}(X)$ it holds that $\varphi \models \varphi \vee \chi$, so $\varphi \succsim \varphi \vee \chi$. Now assume that $\chi \not\models \varphi$. Then we have $MD(\chi) \nsubseteq MD(\varphi)$ and therefore $MD(\varphi \vee \chi) = MD(\varphi) \cup MD(\chi) \neq MD(\varphi)$ which (together with $MD(\varphi) \subseteq MD(\varphi \vee \chi)$) gives $MD(\varphi) \subset MD(\varphi \vee \chi)$, i.e. $\varphi \succ \varphi \vee \chi$.

Now assume that $\varphi \vee \chi$ is not a maximal downward refinement of φ. Let α be a formula such that $\varphi \succ \alpha \succ \varphi \vee \chi$. Then we have $MD(\varphi) \subset MD(\alpha) \subset MD(\varphi \vee \chi)$ and therefore $|MD(\varphi)| < |MD(\alpha)| < |MD(\varphi \vee \chi)|$ since $MD(\varphi)$, $MD(\alpha)$ and $MD(\chi)$ are finite. Since $\chi \in \textsc{MinTerms}(X)$ we have $|MD(\chi)| = 1$, so either $|MD(\alpha)| = |MD(\varphi)|$ or $|MD(\alpha)| = |MD(\varphi)| + 1$. This is a contradiction, so the claim is proved. $\square$

Again we will *collect* all downward refinements of φ as follows:

$$\Psi_d(\varphi) = \{\varphi \vee \chi \mid \chi \in \textsc{MinTerms}(X)\}\,.$$

Example 10.3.1

Consider the set $X = \{p_1, p_2, p_3\}$ and the formula $\varphi = p_1 \rightarrow (p_2 \rightarrow p_3)$ from Example 10.2.1. Here we have

$$\begin{array}{rcl}
\Psi_d(\varphi) & = & \{\varphi \vee \chi \mid \chi \in \textsc{MinTerms}(X)\} \\
& = & \left\{\begin{array}{c}
(p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (\neg p_1 \wedge \neg p_2 \wedge \neg p_3), \\
(p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (\neg p_1 \wedge \neg p_2 \wedge p_3), \\
(p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (\neg p_1 \wedge p_2 \wedge \neg p_3), \\
(p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (\neg p_1 \wedge p_2 \wedge p_3), \\
(p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (p_1 \wedge \neg p_2 \wedge \neg p_3), \\
(p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (p_1 \wedge p_2 \wedge p_3), \\
(p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (p_1 \wedge p_2 \wedge \neg p_3), \\
(p_1 \rightarrow (p_2 \rightarrow p_3)) \wedge (p_1 \wedge p_2 \wedge p_3)
\end{array}\right\}.
\end{array}$$

10.3.2. Formulas with Temporal Operators

In full analogy to the upward case we will define a set $\Gamma_d(\varphi)$ for an LTL–formula φ as follows:

$$\begin{array}{rcl}
\Gamma_d(\varphi) & = & \Psi_d(\varphi) \\
& \cup & \left\{\varphi[\mathsf{F}\chi]_p \mid p \in \textsc{Pos}(\varphi), \varphi|_p = \mathsf{X}^i\chi \text{ for some } i \geq 0\right\} \\
& \cup & \left\{\varphi[\mathsf{X}^i\chi]_p \mid p \in \textsc{Pos}(\varphi), \varphi|_p = \mathsf{G}\chi \text{ for every } i \geq 0\right\} \\
& \cup & \left\{\varphi[\bar{\varphi}_1 \star \varphi_2]_p, \varphi[\varphi_1 \star \bar{\varphi}_2]_p \mid \begin{array}{r} p \in \textsc{Pos}(\varphi), \varphi|_p = \varphi_1 \star \varphi_2, \bar{\varphi}_1 \in \Gamma_d(\varphi_1), \\ \bar{\varphi}_2 \in \Gamma_d(\varphi_2), \star \in \{\mathsf{U}, \mathsf{R}\} \end{array}\right\}.
\end{array}$$

Again we assume that the formulas $\bar{\varphi}_1$ and $\bar{\varphi}_2$ from $\Gamma_d(\varphi_1)$ and $\Gamma_d(\varphi_2)$ satisfy the properties $\varphi_1 \gtrsim \bar{\varphi}_1$ and $\varphi_2 \gtrsim \bar{\varphi}_2$. The properties of $\Gamma_d(\varphi)$ are proved as in the upward case.

Lemma 10.3.1 (Replacement–Lemma, Downward–Version)
Let φ and ψ be LTL–formulas, let $p \in \textsc{Pos}(\varphi)$ be a position in φ. If $\psi \gtrsim \varphi|_p$, then $\varphi \gtrsim \varphi[\psi]_p$.

Proof. analogous to the upward case. $\square$

So we have that the formulas from $\Gamma_d(\varphi)$ are downward refinements of φ.

Theorem 10.3.2
For every LTL–formula φ and every $\psi \in \Gamma_u(\varphi)$ it holds that $\varphi \gtrsim \psi$.

Proof. analogous to the upward case. $\square$

In the following section we will show how the choice of a formula from $\Gamma_u(\varphi)$ or $\Gamma_d(\varphi)$ can be implemented as a manipulation of the representing automaton $\mathcal{A}_\varphi$.

10.4. Modifying Automata by Application of Refinement Operations

10.4.1. Upward Refinement

We will now present algorithms which allow upward–refinement of given Büchi–automata by manipulating the set of states (and updating the transition relation, the set of initial state and the acceptance component). As pointed out in the foregoing section we can distinguish between refinement by application of propositional formulas and refinement by application of temporal formulas.

All algorithms will only differ in the way the new states are constructed. So we will place our attention on the procedures to construct new states from given ones. For the rest of this section we will assume that $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ is given.

Propositional Refinement

Assume that $\textsc{Var}(\varphi) = \{p_1, \ldots, p_{|\textsc{Var}(\varphi)|}\}$ and that $\chi = \bigwedge_{i=1}^{|\textsc{Var}(\varphi)|} l_i \in \textsc{MinTerms}(\textsc{Var}(\varphi))$ for $l_i \in \{p_i, \neg p_i\}$ are given. Then the construction of $\mathcal{A}_{\varphi \wedge \neg \chi}$ is simple: First we have to construct $\mathrm{NNF}(\neg\chi)$. After that we construct $\mathcal{A}_{\mathrm{NNF}(\neg\chi)}$ and return $\mathcal{A}_{\varphi\wedge\neg\chi} = \mathcal{A}_\varphi \times \mathcal{A}_{\mathrm{NNF}(\neg\chi)}$. Since $\chi = \bigwedge_{i=1}^{|\textsc{Var}(\varphi)|} l_i$ we have $\alpha = \mathrm{NNF}(\neg\chi) = \mathrm{NNF}\left(\neg \bigwedge_{i=1}^{|\textsc{Var}(\varphi)|} l_i\right) = \mathrm{NNF}\left(\bigvee_{i=1}^{|\textsc{Var}(\varphi)|} \neg l_i\right) = \bigvee_{i=1}^{|\textsc{Var}(\varphi)|} \neg l_i$. Algorithm 13 gives the implementation of the strategy described here.

Algorithm 13 Propositional Upward Refinement

Input:

- Büchi–automaton $\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$
- $\chi \in \textsc{MinTerms}(\textsc{Var}(\varphi))$.

Output: $\mathcal{A}_{\varphi\wedge\neg\chi}$.

```
1: compute $\alpha = \mathrm{NNF}(\neg\chi)$
2: compute $\mathcal{A}_\alpha$
3: return $\mathcal{A}_\varphi \times \mathcal{A}_\alpha$
```

By soundness of the product operation $\times$ we have the following theorem.

Theorem 10.4.1

Let φ be an $\textsc{Ltl}$–formula, let $\mathcal{A}_\varphi$ and let $\chi \in \textsc{MinTerms}(\varphi)$ be given. Then Algorithm 13 returns $\mathcal{A}_{\varphi\wedge\neg\chi}$.

Temporal Refinement

The next step is now to present a procedure which allows the introduction of temporal operators. We will see that for each of the rewritten formulas from $\Gamma_u(\varphi) \setminus \Psi_u(\varphi)$. So assume that any such $\psi \in \Gamma_u(\varphi) \setminus \Psi_u(\varphi)$ is chosen. We will proceed by distinction of the form of ψ.

Case 1 $\psi = \varphi[\mathsf{X}^i\chi]_p$ for some $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \mathsf{F}\chi$, some $\textsc{Ltl}$–formula χ and some $i \geq 0$. By definition of the set of states of a Büchi–automaton, for every state

s such that $\mathsf{X}^i\chi \in \Phi(s)$ there has to be at least one s' such that $\mathsf{X}^{i-1} \in \Phi(s')$. Our construction will have to take this into account.

Case 2 $\psi = \varphi[\mathsf{G}\chi]_p$ for some $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \mathsf{X}^i\chi$. This case is *simpler* in the sense that some states might be deleted and so the resulting automaton might be smaller than the original one.

Case 3 If $\psi = \varphi[\bar{\varphi}_1 \star \varphi_2]_p$ or $\psi = \varphi[\varphi_1 \star \bar{\varphi}_2]_p$ for $\star \in \{\mathsf{U}, \mathsf{R}\}$ and $p \in \textsc{Pos}(\varphi)$ with $\varphi|_p = \varphi_1 \star \varphi_2$, then states from $\mathcal{A}_\psi$ emerge from states of $\mathcal{A}_\varphi$ and states from $\mathcal{A}_{\bar{\varphi}_1}$ (respectively from $\mathcal{A}_{\bar{\varphi}_2}$) be *merging* the sets of formulas stored in these states. We will see below how this can be achieved.

We will now examine the cases sketched above in more detail.

Case 1 We will first present a method which is suitable for the situation described in case 1. Assume that $p \in \textsc{Pos}(\varphi)$ is given such that $\varphi|_p = \texttt{true}\mathsf{U}\chi \equiv \mathsf{F}\chi$ for some $\textsc{Ltl}$–formula χ and assume that $i \geq 0$ is some fixed integer. If $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ then we have to process every $s = (\{\varphi_1, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\}) \in \textsc{States}$ and check if the following cases occur:

Case 1.1 There is $i_0 \in \{1, \ldots, n\}$ such that $\varphi_{i_0} = \texttt{true}\mathsf{U}\chi$ and $p_i = p$. In this case we have to construct states $\bar{s}_0, \ldots, \bar{s}_i$ as follows:

$$\bar{s}_j = (\{\varphi_1, \ldots, \varphi_{i_0-1}, \mathsf{X}^j\chi, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_{i_0-1}, p_{i_0}1^{i-j}, p_{i_0+1}, \ldots, p_n\}).$$

Here the term 1^{i-j} denotes a sequence of $i-j$ occurrences of the letter 1.

Case 1.2 Case 1.1 is not fulfilled but there is some $i_0 \in \{1, \ldots, n\}$ such that $p_{i_0} < p$ and $\texttt{true}\mathsf{U}\chi \sqsubset \varphi_{i_0}$. In this case the formula to be manipulated is a proper subformula of the formula φ_{i_0} and therefore it has to be replaced. Let $\bar{p}$ be from $\mathbb{N}^*$ such that $p_{i_0}\bar{p} = p$. Then we have to construct the state

$$\bar{s} = (\{\varphi_1, \ldots, \varphi_{i_0-1}, \varphi_{i_0}[\mathsf{X}^i\chi]_{\bar{p}}, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\}).$$

Case 1.3 In the case that neither of the above cases is fulfilled we only add the state s to the set of new states.

The method described here is summarized in Algorithm 14.

Case 2 In contrast to the method presented in the foregoing case constructing the set of states of the automaton $\mathcal{A}_{\varphi[\mathsf{G}\chi]_p}$ given some $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \mathsf{X}^i\chi$ for some $i \geq 0$ might yield a smaller set of states since the states which have been built in order to guarantee that χ holds can be deleted. Again let $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ be given. If $s = (\{\varphi_1, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$ is the state which is actually processed, we have to distinguish the following cases:

Case 2.1 If there is $i_0 \in \{1, \ldots, n\}$ such that $\varphi_{i_0} = \mathsf{X}^i\chi$ and $p_{i_0} = p$, then we can construct *two* new states:

$$\begin{aligned} \bar{s}_1 &= (\{\varphi_1, \ldots, \varphi_{i_0-1}, \mathtt{false}\mathsf{R}\chi, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\}) \text{ and} \\ \bar{s}_2 &= (\{\varphi_1, \ldots, \varphi_{i_0-1}, \chi, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_{i_0-1}, p_{i_0}1, p_{i_0+1}, \ldots, p_n\}). \end{aligned}$$

These states are added to the set of new states. Following the construction of these states we can identify the states which might be deleted. These are the states $s' \neq s$ such that

$$s' = (\{\varphi_1, \ldots, \varphi_{i_0-1}, \mathsf{X}^{i-j}\chi, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_{i_0-1}, p_{i_0}1^j, p_{i_0+1}, \ldots, p_n\}).$$

Each such state is marked as *to be deleted.*

Case 2.2 Case 2.1 is not fulfilled but there is $i_0 \in \{1, \ldots, n\}$ such that $p_{i_0} < p$ and $\mathsf{X}^i\chi$ is a proper subformula of φ_{i_0}. Here we will identify $\bar{p} \in \mathbb{N}^*$ such that $p_{i_0}\bar{p} = p$ and construct the state

$$\bar{s} = (\{\varphi_1, \ldots, \varphi_{i_0-1}, \varphi_{i_0}[\mathtt{false}\mathsf{R}\chi]_{\bar{p}}, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\}).$$

Case 2.3 If neither case 2.1 or case 2.2 is fulfilled, we only add s to the set of new states.

The above construction is formalized in Algorithm 15.

Case 3 Let us now assume that $\psi = \varphi[\bar{\varphi}_1 \star \varphi_2]_p$ for some $\bar{\varphi}_1 \in \Gamma_u(\varphi_1)$, some $\star \in \{\mathsf{U}, \mathsf{R}\}$ and some $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \varphi_1 \star \varphi_2$. Let $\mathcal{A}_{\bar{\varphi}_1} = (\Sigma', \textsc{States}', \delta', S_0', \mathcal{F}')$ be given. Our construction will be divided into three parts.

Step 1 First we will *rename* the positions from the states of $\mathcal{A}_{\bar{\varphi}_1}$ in order to match the positions in ψ. That is if $s' = (\Phi(s'), \{p_1, \ldots, p_n\}) \in \textsc{States}'$ is given, then s has to be changed to $(\Phi(s'), \{p1p_1, \ldots, p1p_n)$.

Step 2 We will then process every $s = \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right) \in \textsc{States}$ as follows:

1. for every $i_0 \in \{1, \ldots, n_1\}$ such that $p_{i_0}^{(1)} = p$ we have to carry out an *explicit replacement*, that is we replace s by

$$\bar{s} = \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{i_0-1}^{(1)}, \varphi_{i_0}^{(1)}[\bar{\varphi}_1]_1, \varphi_{i_0+1}^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right),$$

2. for every $i_0 \in \{1, \ldots, n_1\}$ such that $p_{i_0}^{(1)} < p$ and φ is a proper subformula of $\varphi_{i_0}^{(1)}$ we replace s by

$$\bar{s} = \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{i_0-1}^{(1)}, \varphi_{i_0}^{(1)}[\bar{\varphi}_1]_{\bar{p}1}, \varphi_{i_0+1}^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right)$$

where $\bar{p} \in \mathbb{N}^*$ is such that $p_{i_0}^{(1)}\bar{p} = p$ and

3. for every $i_0 \in \{1, \ldots, n_1\}$ such that $p < p_{i_0}$ we delete the formula $\varphi_{i_0}^{(1)}$ and the associated position $p_{i_0}^{(1)}$. So s is replaced by

$$\bar{s} = \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{i_0-1}^{(1)}, \varphi_{i_0+1}^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{i_0-1}^{(1)}, p_{i_0+1}^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right).$$

Step 3 Assume that after having completed step 2, the state s has the form

$$s = \left(\left\{\varphi_{j_1}^{(1)}, \ldots, \varphi_{j_k}^{(1)}\right\}, \left\{p_{j_1}^{(1)}, \ldots, p_{j_k}^{(1)}\right\}\right).$$

Then for each $s' = \left(\left\{\varphi_1^{(2)}, \ldots, \varphi_{n_2}^{(2)}\right\}, \left\{p_1^{(2)}, \ldots, p_{n_2}^{(2)}\right\}\right) \in \textsc{States}'$ we perform the following actions:

1. $\Phi_{\text{new}} = \left\{\varphi_{j_1}^{(1)}, \ldots, \varphi_{j_k}^{(1)}, \varphi_1^{(2)}, \ldots, \varphi_{n_2}^{(2)}\right\}$,
2. $\textsc{Pos}_{\text{new}} = \left\{p_{j_1}^{(1)}, \ldots, p_{j_k}^{(1)}, p_1^{(2)}, \ldots, p_{n_2}^{(2)}\right\}$ and
3. create the state $s_{\text{new}} = (\Phi_{\text{new}}, \textsc{Pos}_{\text{new}})$ and add it to $\textsc{NewStates}$.

Setting $\Sigma_{\text{new}} = \Sigma \cup \Sigma'$ we can then construct the new transition relation δ', the new set $S_{0,\text{new}}$ of initial states and the new acceptance component $\mathcal{F}'$ as usual and the automaton construction is completed. The algorithm for this construction is given in Algorithm 16.

Case 4 The final case is given by the situation in which $\psi = \varphi[\varphi_1 \star \bar{\varphi}_2]_p$ for some $\bar{\varphi}_2 \in \Gamma_u(\varphi_2)$, some $\star \in \{\mathsf{U}, \mathsf{R}\}$ and some $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \varphi_1 \star \varphi_2$. The construction is then carried out in full analogy to the construction from case 3. So we will only present the algorithm which is depicted in Algorithm 17.

As in the case of propositional upward refinement the above algorithms for constructing the set of new states of the resulting automaton representing the refined formula can be combined with the standard approaches for extracting the initial states, the transition

Algorithm 14 Temporal Upward Refinement: Constructing new States for $\mathsf{F} \mapsto \mathsf{X}^i$

Input:

- $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$
- $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \texttt{true}\mathsf{U}\chi \equiv \mathsf{F}\chi$
- $i \geq 0$

Output: set $\textsc{NewStates}$ of states of $\mathcal{A}_{\varphi[\mathsf{X}^i\chi]_p}$

```
1: $\textsc{NewStates} \leftarrow \{(\emptyset, \emptyset)\}$
2: for each $s \in \textsc{States}$ do
Require: $s = (\{\varphi_1, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$
3:     if there is $i_0$ such that $\varphi_{i_0} = \texttt{true}\mathsf{U}\chi$ and $p_{i_0} = p$ then
4:         for $j = 0, \ldots, i$ do
5:             $\bar{s} \leftarrow (\{\varphi_1, \ldots, \varphi_{i_0-1}, \mathsf{X}^j\chi, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_{i_0-1}, p_{i_0}1^{i-j}, p_{i_0+1}, \ldots, p_n\})$
6:             $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{\bar{s}\}$
7:         end for
8:     else if there is $i_0$ such that $\texttt{true}\mathsf{U}\chi \sqsubset \varphi_{i_0}$ and $p_{i_0} < p$ then
9:         $\bar{p} \leftarrow$ element from $\mathbb{N}^*$ such that $p_{i_0}\bar{p} = p$ $\bar{s} \leftarrow$
           $(\{\varphi_1, \ldots, \varphi_{i_0-1}, \varphi_{i_0}[\mathsf{X}^i\chi]_{\bar{p}}, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$
10:        $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{\bar{s}\}$
11:    else
12:        $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{s\}$
13:    end if
14: end for
```

Algorithm 15 Temporal Upward Refinement: Constructing new States for $\mathsf{X}^i \mapsto \mathsf{G}$

Input:

- $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$
- $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \mathsf{X}^i\chi$

Output: set $\textsc{NewStates}$ of states of $\mathcal{A}_{\varphi[\mathsf{G}\chi]_p}$

```
1: mark each $s \in \textsc{States}$ as *not to be deleted*
2: $\textsc{NewStates} \leftarrow \{(\emptyset, \emptyset)\}$
3: **for** each $s \in \textsc{States}$ which is marked as *not to be deleted* **do**
**Require:** $s = (\{\varphi_1, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$
4:   **if** there is $i_0$ such that $\varphi_{i_0} = \mathsf{X}^i\chi$ and $p_{i_0} = p$ **then**
5:     $\bar{s}_1 \leftarrow (\{\varphi_1, \ldots, \varphi_{i_0-1}, \mathtt{false}\mathsf{R}\chi, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$
6:     $\bar{s}_2 \leftarrow (\{\varphi_1, \ldots, \varphi_{i_0-1}, \chi, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_{i_0-1}, p_{i_0}1, p_{i_0+1}, \ldots, p_n\})$
7:     $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{\bar{s}_1, \bar{s}_2\}$
8:     **for** each $s' \in \textsc{States}$ **do**
**Require:** $s' = (\{\varphi'_1, \ldots, \varphi'_m\}, \{p'_1, \ldots, p'_m\})$
9:       **if** $n = m$ **then**
10:        **if** $\{\varphi'_1, \ldots, \varphi'_m\} = \{\varphi_1, \ldots, \varphi_{i_0-1}, \mathsf{X}^{i-j}\chi, \varphi_{i_0+1}, \ldots, \varphi_n\}$ for some $j \geq 0$
           **then**
11:          **if** $\{p'_1, \ldots, p'_m\} = \{p_1, \ldots, p_{i_0-1}, p_{i_0}1^j, p_{i_0+1}, \ldots, p_n\}$ **then**
12:            mark $s'$ as *to be deleted*
13:          **end if**
14:        **end if**
15:      **end if**
16:    **end for**
17:  **else if** there if $i_0$ such that $p_{i_0} < p$ and $\mathsf{X}^i\chi \sqsubset \varphi_{i_0}$ **then**
18:    $\bar{p} \leftarrow$ element from $\mathbb{N}^*$ such that $p_{i_0}\bar{p} = p$
19:    $\bar{s} \leftarrow (\{\varphi_1, \ldots, \varphi_{i_0-1}, \varphi_{i_0}[\mathtt{false}\mathsf{R}\chi]_{\bar{p}}, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$
20:    $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{\bar{s}\}$
21:  **else**
22:    $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{s\}$
23:  **end if**
24: **end for**
25: **return** $\textsc{NewStates}$
```

Algorithm 16 Temporal Upward Refinement: Constructing new States for Replacement of Eventualities (replacing the first component)

Input:

- $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$
- $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \varphi_1 \star \varphi_2$ for $\star \in \{\mathsf{U}, \mathsf{R}\}$
- $\mathcal{A}_{\overline{\varphi}} = (\Sigma', \textsc{States}', \delta', S'_0, \mathcal{F}')$ for some $\overline{\varphi} \in \Gamma_u(\varphi_1)$

Output: set $\textsc{NewStates}$ of states of $\mathcal{A}_{\varphi[\bar{\varphi}\star\varphi_2]_p}$

```
1: for each $s' \in \textsc{States}'$ do
Require: $s' = (\Phi(s'), \{p'_1, \ldots, p'_n\})$
2:   $s \leftarrow (\Phi(s'), \{pp'_1, \ldots, pp'_n\})$
3: end for
4: $\textsc{NewStates} \leftarrow \{(\emptyset, \emptyset)\}$
5: for each $s \in \textsc{States}$ do
Require: $s = \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right)$
6:   for $i_0 = 1, \ldots, n_1$ do
7:     if $p_{i_0}^{(1)} \leq p$ and $\varphi_1$ is a subformula of $\varphi_{i_0}^{(1)}$ then
8:       $\bar{p} \leftarrow$ element from $\mathbb{N}^*$ such that $p_{i_0}^{(1)}\bar{p} = p$
9:       $s \leftarrow \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{i_0-1}^{(1)}, \varphi_{i_0}^{(1)}[\overline{\varphi}]_{\bar{p}1}, \varphi_{i_0+1}^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right)$
10:    else if $p < p_{i_0}^{(1)}$ then
11:      $s \leftarrow \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{i_0-1}^{(1)}, \varphi_{i_0+1}^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{i_0-1}^{(1)}, p_{i_0+1}^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right)$
12:    end if
13:  end for
Require: $s = \left(\{\varphi_{j_1}^{(1)}, \ldots, \varphi_{j_k}^{(1)}\}, \{p_{j_1}^{(1)}, \ldots, p_{j_k}^{(1)}\}\right)$
14:  for each $s' \in \textsc{States}'$ do
Require: $s' = \left(\left\{\varphi_1^{(2)}, \ldots, \varphi_{n_2}^{(2)}\right\}, \left\{p_1^{(2)}, \ldots, p_{n_2}^{(2)}\right\}\right)$
15:    $\Phi_{\text{new}} \leftarrow \left\{\varphi_{j_1}^{(1)}, \ldots, \varphi_{j_k}^{(1)}, \varphi_1^{(2)}, \ldots, \varphi_{n_2}^{(2)}\right\}$
16:    $\textsc{Pos}_{\text{new}} \leftarrow \left\{p_{j_1}^{(1)}, \ldots, p_{j_k}^{(1)}, p_1^{(2)}, \ldots, p_{n_2}^{(2)}\right\}$
17:    $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{(\Phi_{\text{new}}, \textsc{Pos}_{\text{new}})\}$
18:  end for
19: end for
20: return $\textsc{NewStates}$
```

Algorithm 17 Temporal Upward Refinement: Constructing new States for Replacement of Eventualities (replacing the second component)

Input:

- $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$
- $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \varphi_1 \star \varphi_2$ for $\star \in \{\mathsf{U}, \mathsf{R}\}$
- $\mathcal{A}_{\overline{\varphi}} = (\Sigma', \textsc{States}', \delta', S_0', \mathcal{F}')$ for some $\overline{\varphi} \in \Gamma_u(\varphi_1)$

Output: set $\textsc{NewStates}$ of states of $\mathcal{A}_{\varphi[\varphi_1 \star \bar{\varphi}]_p}$

```
1: for each $s' \in \textsc{States}'$ do
Require: $s' = (\Phi(s'), \{p'_1, \ldots, p'_n\})$
2:   $s \leftarrow (\Phi(s'), \{pp'_1, \ldots, pp'_n\})$
3: end for
4: $\textsc{NewStates} \leftarrow \{(\emptyset, \emptyset)\}$
5: for each $s \in \textsc{States}$ do
Require: $s = \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right)$
6:   for $i_0 = 1, \ldots, n_1$ do
7:     if $p_{i_0}^{(1)} \leq p$ and $\varphi_1$ is a subformula of $\varphi_{i_0}^{(1)}$ then
8:       $\bar{p} \leftarrow$ element from $\mathbb{N}^*$ such that $p_{i_0}^{(1)}\bar{p} = p$
9:       $s \leftarrow \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{i_0-1}^{(1)}, \varphi_{i_0}^{(1)}[\overline{\varphi}]_{\bar{p}2}, \varphi_{i_0+1}^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right)$
10:    else if $p < p_{i_0}^{(1)}$ then
11:      $s \leftarrow \left(\left\{\varphi_1^{(1)}, \ldots, \varphi_{i_0-1}^{(1)}, \varphi_{i_0+1}^{(1)}, \ldots, \varphi_{n_1}^{(1)}\right\}, \left\{p_1^{(1)}, \ldots, p_{i_0-1}^{(1)}, p_{i_0+1}^{(1)}, \ldots, p_{n_1}^{(1)}\right\}\right)$
12:    end if
13:  end for
Require: $s = \left(\{\varphi_{j_1}^{(1)}, \ldots, \varphi_{j_k}^{(1)}\}, \{p_{j_1}^{(1)}, \ldots, p_{j_k}^{(1)}\}\right)$
14:  for each $s' \in \textsc{States}'$ do
Require: $s' = \left(\left\{\varphi_1^{(2)}, \ldots, \varphi_{n_2}^{(2)}\right\}, \left\{p_1^{(2)}, \ldots, p_{n_2}^{(2)}\right\}\right)$
15:    $\Phi_{\text{new}} \leftarrow \left\{\varphi_{j_1}^{(1)}, \ldots, \varphi_{j_k}^{(1)}, \varphi_1^{(2)}, \ldots, \varphi_{n_2}^{(2)}\right\}$
16:    $\textsc{Pos}_{\text{new}} \leftarrow \left\{p_{j_1}^{(1)}, \ldots, p_{j_k}^{(1)}, p_1^{(2)}, \ldots, p_{n_2}^{(2)}\right\}$
17:    $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{(\Phi_{\text{new}}, \textsc{Pos}_{\text{new}})\}$
18:  end for
19: end for
20: return $\textsc{NewStates}$
```

function and the acceptance component of the resulting automaton. A procedure of how to achieve this is straightforward to implement. Therefore we will not give it in detail but will merely concentrate on the following theorem.

Theorem 10.4.2

Let $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ be an automaton representing an Ltl–formula φ and let $\psi \in \Gamma_u(\varphi)$ be an upward–refinement of φ. Then the sets of states computed by Algorithms 14, 15, 16 and 17 are correct.

Proof. Let an Ltl–formula φ be given and assume that some $\psi \in \Gamma_u(\varphi)$ is chosen. Assume that $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ is given (and is constructed *correctly*). Let $\textsc{States}'$ be the set of states which has been constructed by application of one of the algorithms presented above and let $\textsc{States}_{cor}$ be the *correct* set of states of $\mathcal{A}_\psi$. We will have to prove that $\textsc{States}' = \textsc{States}_{cor}$. The direction $\subseteq$ is simple: Let $s = (\Phi(s), \textsc{Pos})$ be any state from $\textsc{States}'$. By assumption that $\mathcal{A}_\varphi$ is constructed correctly we have that `false` $\notin \Phi(s)$ and since the algorithms presented above do not introduce conjunctions or disjunctions we have that if $\varphi_1 \wedge \varphi_2 \in \Phi(s)$ $(\varphi_1 \vee \varphi_2 \in \Phi(s))$ then $\varphi_1 \in \Phi(s)$ and (or) $\varphi_2 \in \Phi(s)$. The correctness of the positions stored in $\textsc{Pos}$ is immediate. So the direction $\subseteq$ is proved.

Now assume that $s = (\Phi(s), \textsc{Pos}) = (\{\varphi_1, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$ is a state in $\textsc{States}_{cor}$. If $p \notin \textsc{Pos}$, then we trivially have that $s \in \textsc{States}'$ since then $s \in \textsc{States}$. If $p \in \textsc{Pos}$, say $p = p_j$ then the claim is proven by showing that all necessary dependencies have been constructed. These dependencies have the form $\mathsf{X}^i\chi$, that is if they are constructed then $\mathsf{X}^{i-1}\chi, \ldots, \mathsf{X}^1\chi, \mathsf{X}^0\chi$ have to be constructed and included. But this is done by the algorithms so $s \in \textsc{States}'$ and the direction $\supseteq$ is proved. $\square$

Since the extraction of the remaining components of $\mathcal{A}_\psi$ is more or less trivial, we have that the refinement procedures described are correct.

10.4.2. Downward Refinement

Propositional Refinement

As in the case of propositional upward refinement we can define an algorithm for constructing the refined formula $\varphi \vee \chi$ from a formula φ, the automaton $\mathcal{A}_\varphi$ and some $\chi \in \textsc{MinTerms}(\textsc{Var}(\varphi))$ by first constructing the automaton $\mathcal{A}_\chi$ and then constructing the automaton $\mathcal{A}_{\varphi\vee\chi} = \mathcal{A}_\varphi || \mathcal{A}_\chi$. Since the operation $||$ is sound we immediately have the soundness of the refinement procedure. The procedure itself is given in Algorithm 18

Theorem 10.4.3

Let φ be an Ltl–formula, let $\mathcal{A}_\varphi$ and let $\chi \in \textsc{MinTerms}(\varphi)$ be given. Then Algorithm 18 returns $\mathcal{A}_{\varphi\vee\chi}$.

Algorithm 18 Propositional Downward Refinement

Input:

- Büchi–automaton $\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$
- $\chi \in \textsc{MinTerms}(\textsc{Var}(\varphi))$.

Output: $\mathcal{A}_{\varphi\vee\chi}$.

```
1: compute $\alpha = \text{NNF}(\chi)$
2: compute $\mathcal{A}_\alpha$
3: **return** $\mathcal{A}_\varphi || \mathcal{A}_\alpha$
```

Temporal Refinement

The case of refining Ltl–formulas by adding respectively changing temporal operators has been described for the case of upward refinement in chapter 10.4.1. Downward refinement is more or less dual to upward refinement as we will see soon. Again we can assume that some Ltl–formula φ and some element $\psi \in \Gamma_d(\varphi) \setminus \Psi_d(\varphi)$ are fixed. As in chapter 10.4.1 we have to distinguish the following cases for ψ:

Case 1 $\psi = \varphi[\mathsf{X}^i\chi]_p$ for some $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \mathsf{G}\chi$ and some $i \geq 0$.

This case can be seen as the inversion of case 2 from chapter 10.4.1 where certain states had been marked as *to be deleted.* Consequently the states which are deleted there have to be introduced here. So assume that any element $s = (\{\varphi_1, \dots, \varphi_n\}, \{p_1, \dots, p_n\}) \in \textsc{States}$ is given. We can identify the following subcases:

Case 1.1 If there is $i_0 \in \{1, \dots, n\}$ such that $\varphi_{i_0} = \texttt{false}\mathsf{R}\chi$ and $p_{i_0} = p$, then we have to construct $i + 1$ states $\bar{s}_0, \dots, \bar{s}_i$ as follows:

$$\bar{s}_j = (\{\varphi_1, \dots, \varphi_{i_0-1}, \mathsf{X}^j\chi, \varphi_{i_0+1}, \dots, \varphi_n\}, \{p_1, \dots, p_{i_0-1}, p_{i_0}1^{i-j}, p_{i_0+1}, \dots, p_n).$$

Case 1.2 Case 1.1 is not fulfilled but there is $i_0 \in \{1, \dots, n\}$ such that $p_{i_0} < p$ and $\texttt{false}\mathsf{R}\chi$ is a proper subformula of φ_{i_0}. Then let $\bar{p} \in \mathbb{N}^*$ be such that $p_{i_0}\bar{p} = p$. We construct the state

$$\bar{s} = (\{\varphi_1, \dots, \varphi_{i_0-1}, \varphi_{i_0}[\mathsf{X}^i\chi]_{\bar{p}}, \varphi_{i_0+1}, \dots, \varphi_n\}, \{p_1, \dots, p_n\}).$$

Case 1.3 If neither of the above cases is fulfilled we add the original state s to $\textsc{NewStates}$.

The complete procedure is given in Algorithm 19.

Case 2 By analogy to the case of constructing $\mathcal{A}_{\varphi[\mathsf{G}\chi]_p}$ where $\mathsf{X}^i\chi$ had been replaced by $\texttt{false}\mathsf{R}\chi$ we can construct $\mathcal{A}_{\varphi[\mathsf{F}\chi]_p}$ from $\mathcal{A}_\varphi$, $i \geq 0$ and some $p \in \textsc{Pos}(\varphi)$ where $\varphi|_p = \mathsf{X}^i\chi$. If any $s = (\{\varphi_1, \dots, \varphi_n\}, \{p_1, \dots, p_n\}) \in \textsc{States}$ is processed, then we distinguish the following cases:

Case 2.1 There is $i_0 \in \{1, \dots, n\}$ such that $\varphi_{i_0} = \mathsf{X}^i\chi$ and $p_{i_0} = p$. Then we construct the following states:

$$\bar{s}_1 \quad = \quad (\{\varphi_1, \dots, \varphi_{i_0-1}, \texttt{true}\mathsf{U}\chi, \varphi_{i_0+1}, \dots, \varphi_n\}, \{p_1, \dots, p_n\}) \text{ and}$$

$$\bar{s}_2 \;=\; (\{\varphi_1,\dots,\varphi_{i_0-1},\chi,\varphi_{i_0+1},\dots,\varphi_n\},\{p_1,\dots,p_{i_0-1},p_{i_0}1,p_{i_0+1},\dots,p_n\}).$$

After $\bar{s}_1$ and $\bar{s}_2$ have been added to NewStates, we mark each $s' \neq s$ as *to be deleted* (see chapter 10.4.1) which have the form

$$s' = (\{\varphi_1,\dots,\varphi_{i_0-1},\mathsf{X}^{i-j}\chi,\varphi_{i_0+1},\dots,\varphi_n\},\{p_1,\dots,p_{i_0-1},p_{i_0}1^j,p_{i_0+1},\dots,p_n\})$$

for some $j \geq 0$.

Case 2.2 If case 2.1 is not fulfilled but there is $i_0 \in \{1,\dots,n\}$ such that $p_{i_0} < p$ and $\mathsf{X}^i\chi$ is a proper subformula of φ_{i_0} we identify $\bar{p} \in \mathbb{N}^*$ such that $p_{i_0}\bar{p} = p$ and construct

$$\bar{s} = (\{\varphi_1,\dots,\varphi_{i_0-1},\varphi_{i_0}[\mathtt{true}\mathsf{U}\chi]_{\bar{p}},\varphi_{i_0+1},\dots,\varphi_n\},\{p_1,\dots,p_n\})$$

and add $\bar{s}$ to NewStates.

Case 2.3 Neither of the above cases is fulfilled. then s is added to the set NewStates.

The complete procedure is summarized in Algorithm 20.

Case 3 If the refined formula is $\psi = \varphi[\bar{\varphi}_1 \star \varphi_2]_p$ for some $p \in \mathrm{Pos}(\varphi)$ such that $\varphi|_p = \varphi_1 \star \varphi_2$, some $\star \in \{\mathsf{U},\mathsf{R}\}$ and some $\bar{\varphi}_1 \in \Gamma_d(\varphi_1)$, then we can re–use Algorithm 16 *without any changes* since this algorithm only refers to $\bar{\varphi}_1$ and not to membership of $\bar{\varphi}_1$ in $\Gamma_u(\varphi_1)$ or $\Gamma_d(\varphi_2)$.

Case 4 By analogy to the foregoing case we can re–use Algorithm 17 in order to compute the set of states for $\mathcal{A}_\psi$ in the case that $\psi = \varphi[\varphi_1 \star \bar{\varphi}_2]_p$ for $p \in \mathrm{Pos}(\varphi_2)$ with $\varphi|_p = \varphi_1 \star \varphi_2$, $\star \in \{\mathsf{U},\mathsf{R}\}$ and $\bar{\varphi}_2 \in \Gamma_d(\varphi_2)$.

So all possible cases for the elements $\psi \in \Gamma_d(\varphi)$ are covered.

Algorithm 19 Temporal Downward Refinement: Constructing new States for $\mathsf{G} \mapsto \mathsf{X}^i$

Input:

- $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$
- $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \texttt{false}\mathsf{R}\chi \equiv \mathsf{G}\chi$
- $i \geq 0$

Output: set $\textsc{NewStates}$ of states of $\mathcal{A}_{\varphi[\mathsf{X}^i\chi]_p}$

```
 1: $\textsc{NewStates} \leftarrow \{(\emptyset, \emptyset)\}$
 2: for each $s \in \textsc{States}$ do
Require: $s = (\{\varphi_1, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$
 3:   if there is $i_0$ such that $\varphi_{i_0} = \texttt{false}\mathsf{R}\chi$ and $p_{i_0} = p$ then
 4:     for $j = 0, \ldots, i$ do
 5:       $\bar{s} \leftarrow (\{\varphi_1, \ldots, \varphi_{i_0-1}, \mathsf{X}^j\chi, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_{i_0-1}, p_{i_0}1^{i-j}, p_{i_0+1}, \ldots, p_n\})$
 6:       $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{\bar{s}\}$
 7:     end for
 8:   else if there is $i_0$ such that $\texttt{false}\mathsf{R}\chi \sqsubset \varphi_{i_0}$ and $p_{i_0} < p$ then
 9:     $\bar{p} \leftarrow$ element from $\mathbb{N}^*$ such that $p_{i_0}\bar{p} = p$ $\bar{s} \leftarrow$
        $(\{\varphi_1, \ldots, \varphi_{i_0-1}, \varphi_{i_0}[\mathsf{X}^i\chi]_{\bar{p}}, \varphi_{i_0+1}, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$
10:     $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{\bar{s}\}$
11:   else
12:     $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{s\}$
13:   end if
14: end for
```

Algorithm 20 Temporal Downward Refinement: Constructing new States for $\mathsf{X}^i \mapsto \mathsf{F}$

Input:

- $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$
- $p \in \textsc{Pos}(\varphi)$ such that $\varphi|_p = \mathsf{X}^i\chi$

Output: set $\textsc{NewStates}$ of states of $\mathcal{A}_{\varphi[\mathsf{F}\chi]_p}$

```
1: mark each $s \in \textsc{States}$ as *not to be deleted*
2: $\textsc{NewStates} \leftarrow \{(\emptyset, \emptyset)\}$
3: **for** each $s \in \textsc{States}$ which is marked as *not to be deleted* **do**
**Require:**  $s = (\{\varphi_1, \dots, \varphi_n\}, \{p_1, \dots, p_n\})$
4:   **if** there is $i_0$ such that $\varphi_{i_0} = \mathsf{X}^i\chi$ and $p_{i_0} = p$ **then**
5:     $\bar{s}_1 \leftarrow (\{\varphi_1, \dots, \varphi_{i_0-1}, \mathtt{true}\mathsf{U}\chi, \varphi_{i_0+1}, \dots, \varphi_n\}, \{p_1, \dots, p_n\})$
6:     $\bar{s}_2 \leftarrow (\{\varphi_1, \dots, \varphi_{i_0-1}, \chi, \varphi_{i_0+1}, \dots, \varphi_n\}, \{p_1, \dots, p_{i_0-1}, p_{i_0}1, p_{i_0+1}, \dots, p_n\})$
7:     $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{\bar{s}_1, \bar{s}_2\}$
8:     **for** each $s' \in \textsc{States}$ **do**
**Require:**    $s' = (\{\varphi'_1, \dots, \varphi'_m\}, \{p'_1, \dots, p'_m\})$
9:       **if** $n = m$ **then**
10:        **if** $\{\varphi'_1, \dots, \varphi'_m\} = \{\varphi_1, \dots, \varphi_{i_0-1}, \mathsf{X}^{i-j}\chi, \varphi_{i_0+1}, \dots, \varphi_n\}$ for some $j \geq 0$
           **then**
11:          **if** $\{p'_1, \dots, p'_m\} = \{p_1, \dots, p_{i_0-1}, p_{i_0}1, p_{i_0+1}, \dots, p_n\}$ **then**
12:            mark $s'$ as *to be deleted*
13:          **end if**
14:        **end if**
15:      **end if**
16:    **end for**
17:  **else if** there if $i_0$ such that $p_{i_0} < p$ and $\mathsf{X}^i\chi \sqsubset \varphi_{i_0}$ **then**
18:    $\bar{p} \leftarrow$ element from $\mathbb{N}^*$ such that $p_{i_0}\bar{p} = p$
19:    $\bar{s} \leftarrow (\{\varphi_1, \dots, \varphi_{i_0-1}, \varphi_{i_0}[\mathtt{true}\mathsf{U}\chi]_{\bar{p}}, \varphi_{i_0+1}, \dots, \varphi_n\}, \{p_1, \dots, p_n\})$
20:    $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{\bar{s}\}$
21:  **else**
22:    $\textsc{NewStates} \leftarrow \textsc{NewStates} \cup \{s\}$
23:  **end if**
24: **end for**
25: **return** $\textsc{NewStates}$
```

By analogy to the situation of constructing the states for the upward–refined automata, we are now able to present the following theorem.

Theorem 10.4.4

Let $\mathcal{A}_\varphi = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ be an automaton representing an Ltl–formula φ and let $\psi \in \Gamma_d(\varphi)$ be an upward–refinement of φ. Then the sets of states computed by Algorithms 19 and 20, are correct.

Proof. Exactly as in the case of upward refinement. □

10.4.3. Extracting Formulas

The final step in the process of refining Ltl–formulas by application of the automata manipulations introduced in this chapter is to extract the refined program which has been *learned*. This is a simple task due to the properties of the automata construction used to build the representing automata.

Recall that the set of initial states S_0 of an automaton $\mathcal{A}_\varphi$ representing φ has been defined as the set of all states s such that $\varphi \in \Phi(s)$. So after having completed the refinement process it suffices to extract this formula from *one* of the initial states of the resulting automaton.

Definition 10.4.1

Let φ_1 and φ_2 be Ltl–formulas. Then $\varphi_1 \sqsubseteq \varphi_2$ if there is a position $p \in \textsc{Pos}(\varphi_2)$ such that $\varphi_2|_p = \varphi_1$. Furthermore we will write $\varphi_1 \sqsubset \varphi_2$ if $\varphi_1 \sqsubseteq \varphi_2$ and $p \neq \varepsilon$.

In other words, $\varphi_1 \sqsubseteq \varphi_2$ ($\varphi_1 \sqsubset \varphi_2$) if and only if φ_1 is a (proper) subformula of φ_2. Using the notation $\sqsubseteq$ we can define the *maximum* of a state s.

Definition 10.4.2

Let $\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ be an extended Büchi–automaton and let $s \in \textsc{States}$.

Then a formula $\varphi \in \Phi(s)$ is called a *maximum* of s if there is no formula $\psi \in \Phi(s)$ such that $\varphi \sqsubset \psi$.

Obviously for every $s \in \textsc{States}$ there is exactly one maximum. This formula will be denoted as $\max(s)$. This enables the extraction of a formula from a Büchi–automaton simply by extracting the maximum $\max(s_0)$ for *any* initial state.

Theorem 10.4.5
Let $\mathcal{A} = (\Sigma, \textsc{States}, \delta, S_0, \mathcal{F})$ be an extended Büchi–automaton and let $s_0 \in S_0$ be any initial state of $\mathcal{A}$. Then if $\mathcal{A} = \mathcal{A}_\varphi$, then $\varphi = \max(s_0)$.

Proof. Immediately by definition of $\mathcal{A}_\varphi$. □

Since we have defined our version of Büchi–automata in such a way that not only the formulas but also the positions of these formulas in the original formula from which the automaton had been constructed are stored, we can extract the maximum of any $s \in S_0$ simply by searching for the position ε in the position–component of s. So let $s = (\{\varphi_1, \ldots, \varphi_n\}, \{p_1, \ldots, p_n\})$ be given and let i_0 be such that $p_{i_0} = \varepsilon$. Then $\varphi_{i_0} = \max(s)$. So searching for the maximum of s can be achieved in time $\mathcal{O}(|\Phi(s)|)$.

10.5. The Identification Process

During the foregoing sections we have developed techniques which are necessary in order to identify Ltl–programs from positive and negative examples. So this final section of this chapter will deal with the topic of how to combine these techniques in order to derive an identification procedure. In order to implement an identification procedure we have to ensure that at every point of time there is a uniquely determined continuation of the refinement process. In other words we have to ensure that our method is deterministic.

So assume that there is a (heuristic) function $\mathfrak{h}$ selecting a type of refinement step to be carried out. Depending on the outcome of the computation carried out by $\mathfrak{h}$ we have to call one of the algorithms developed in the foregoing sections. Formally $\mathfrak{h}$ has to return the following information:

- a number of an algorithm to be called and
- the input data for this algorithm.

Considering the different signatures which Algorithms 13 to 20 have, the following data might be necessary:

- a formula $\chi \in \textsc{MinTerms}(\textsc{Var}(\varphi))$ as input for Algorithms 13 and 18,
- a position $p \in \textsc{Pos}(\varphi)$,
- a value $i \geq 0$ for Algorithms 14 and 19,
- an index $j \in \{1, 2\}$ for Algorithms 16 and 17 selecting which component has to be replaced and
- an automaton $\mathcal{A}_{\overline{\varphi}}$ as input for Algorithms 16 and 17.

So if $\mathfrak{Aut}$ denotes the set of *all* Büchi–automata constructed in the way developed in this and the foregoing chapter, $\mathfrak{h}$ should have the following signature:

$$\mathfrak{h} : \mathfrak{Aut} \to \mathbb{Z}_{10} \times \textsc{MinTerms}(\textsc{Var}(\varphi)) \times \textsc{Pos}(\varphi) \times \mathbb{N} \times \{1, 2\}.$$

For the sake of formal clearness we assume that the components of $\mathfrak{h}(\mathcal{A})$ given any automaton $\mathcal{A} \in \mathfrak{Aut}$ can be accessed by application of a simple projection. That is the projections

$$\begin{aligned}
(\cdot)_1 \quad &: \quad \mathbb{Z}_{10} \times \textsc{MinTerms}(\textsc{Var}(\varphi)) \times \textsc{Pos}(\varphi) \times \mathbb{N} \times \{1,2\} \to \mathbb{Z}_{10}, \\
(\cdot)_2 \quad &: \quad \mathbb{Z}_{10} \times \textsc{MinTerms}(\textsc{Var}(\varphi)) \times \textsc{Pos}(\varphi) \times \mathbb{N} \times \{1,2\} \to \textsc{MinTerms}(\textsc{Var}(\varphi)), \\
(\cdot)_3 \quad &: \quad \mathbb{Z}_{10} \times \textsc{MinTerms}(\textsc{Var}(\varphi)) \times \textsc{Pos}(\varphi) \times \mathbb{N} \times \{1,2\} \to \textsc{Pos}(\varphi),
\end{aligned}$$

$$(\cdot)_4 \quad : \quad \mathbb{Z}_{10} \times \textsc{MinTerms}(\textsc{Var}(\varphi)) \times \textsc{Pos}(\varphi) \times \mathbb{N} \times \{1,2\} \to \mathbb{N} \text{ and}$$
$$(\cdot)_5 \quad : \quad \mathbb{Z}_{10} \times \textsc{MinTerms}(\textsc{Var}(\varphi)) \times \textsc{Pos}(\varphi) \times \mathbb{N} \times \{1,2\} \to \{1,2\}$$

are defined.

Additionally we assume that another function $\bar{\mathfrak{h}} : \mathbb{Z}_2 \times \{1,2\} \times \textsc{Pos}(\varphi) \times \mathfrak{Aut} \to \mathfrak{Aut}$ is given which returns an automaton representing the formula to be inserted during a run of Algorithms 16 and 17. The first argument of $\bar{\mathfrak{h}}$ denotes the direction which the refinement step has to take (0 for upward refinement and 1 for downward refinement) while the second argument denotes the component to be replaced in the formula at the position given by the third component.

So the identification process can be sketched as follows:

Step 0 Given input φ, construct the automaton $\mathcal{A} = \mathcal{A}_\varphi$.

Step 1 Now assume that $\mathcal{E}^+ = \{\varphi_1, \ldots, \varphi_k\}$. For i ranging from 1 to k perform the following loop.

Step 1.1 Construct $\mathcal{A}_{\mathrm{NNF}(\neg\varphi_i)}$,

Step 1.2 construct $\mathcal{A} \times \mathcal{A}_{\mathrm{NNF}(\neg\varphi_i)}$ and

Step 1.3 as long as $L(\mathcal{A} \times \mathcal{A}_{\mathrm{NNF}(\neg\varphi_i)}) \neq \emptyset$ do

Step 1.3.1 compute $\mathfrak{h}(\mathcal{A}) = ((\mathfrak{h}(\mathcal{A}))_1, \ldots, (\mathfrak{h}(\mathcal{A}))_5)$ and

Step 1.3.2 depending on the value of $(\mathfrak{h}(\mathcal{A}))_1$ perform the following actions:

1. if $(\mathfrak{h}(\mathcal{A}))_1 = 0$, then replace $\mathcal{A}$ by the result of Algorithm 13 given inputs $\mathcal{A}$ and $(\mathfrak{h}(\mathcal{A}))_2$,
2. if $(\mathfrak{h}(\mathcal{A}))_1 = 1$, then
   a) compute the set NewStates by application of Algorithm 14 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $(\mathfrak{h}(\mathcal{A}))_4$,
   b) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

and then

c) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

3. if $(\mathfrak{h}(\mathcal{A}))_1 = 2$, then

a) compute the set NewStates by application of Algorithm 15 given inputs $\mathcal{A}$ and $(\mathfrak{h}(\mathcal{A}))_3$,

b) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

and then

c) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

4. if $(\mathfrak{h}(\mathcal{A}))_1 = 3$, then

a) compute the automaton $\bar{\mathcal{A}} = \bar{\mathfrak{h}}(\mathcal{A}, 0, 1, (\mathfrak{h}(\mathcal{A}))_3)$,

b) compute the set NewStates by application of Algorithm 16 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $\bar{\mathcal{A}}$,

c) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NEWSTATES and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NEWSTATES as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NEWSTATES

and then

d) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

5. if $(\mathfrak{h}(\mathcal{A}))_1 = 4$, then

a) compute the automaton $\bar{\mathcal{A}} = \bar{\mathfrak{h}}(\mathcal{A}, 0, 2, (\mathfrak{h}(\mathcal{A}))_3)$,

b) compute the set NEWSTATES by application of Algorithm 17 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $\bar{\mathcal{A}}$,

c) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NEWSTATES and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NEWSTATES as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NEWSTATES

and then

d) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

6. if $(\mathfrak{h}(\mathcal{A}))_1 = 5$, then replace $\mathcal{A}$ by the result of Algorithm 18 given inputs $\mathcal{A}$ and $(\mathfrak{h}(\mathcal{A}))_2$,

7. if $(\mathfrak{h}(\mathcal{A}))_1 = 6$, then

a) compute the set NEWSTATES by application of Algorithm 19 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $(\mathfrak{h}(\mathcal{A}))_4$,

b) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

and then

c) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

8. if $(\mathfrak{h}(\mathcal{A}))_1 = 7$, then

a) compute the set NewStates by application of Algorithm 20 given inputs $\mathcal{A}$ and $(\mathfrak{h}(\mathcal{A}))_3$,

b) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

and then

c) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

9. if $(\mathfrak{h}(\mathcal{A}))_1 = 8$, then

a) compute the automaton $\bar{\mathcal{A}} = \bar{\mathfrak{h}}(\mathcal{A}, 1, 1, (\mathfrak{h}(\mathcal{A}))_3)$,

b) compute the set NewStates by application of Algorithm 16 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $\bar{\mathcal{A}}$,

c) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NEWSTATES and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NEWSTATES as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NEWSTATES

and then

d) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

and

10. if $(\mathfrak{h}(\mathcal{A}))_1 = 9$, then

a) compute the automaton $\bar{\mathcal{A}} = \bar{\mathfrak{h}}(\mathcal{A}, 1, 2, (\mathfrak{h}(\mathcal{A}))_3)$,

b) compute the set NEWSTATES by application of Algorithm 17 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $\bar{\mathcal{A}}$,

c) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NEWSTATES and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NEWSTATES as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NEWSTATES

and then

d) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$.

Step 2 Now assume that $\mathcal{E}^- = \{\varphi_1, \dots, \varphi_l\}$. For i ranging from 1 to l perform the following loop.

Step 2.1 Construct $\mathcal{A}_{\text{NNF}(\neg\varphi_i)}$,

Step 2.2 construct $\mathcal{A} \times \mathcal{A}_{\mathrm{NNF}(\neg\varphi_i)}$ and

Step 2.3 as long as $L(\mathcal{A} \times \mathcal{A}_{\mathrm{NNF}(\neg\varphi_i)}) = \emptyset$ do

Step 2.3.1 compute $\mathfrak{h}(\mathcal{A}) = ((\mathfrak{h}(\mathcal{A}))_1, \ldots, (\mathfrak{h}(\mathcal{A}))_5)$ and

Step 2.3.2 depending on the value of $(\mathfrak{h}(\mathcal{A}))_1$ perform the following actions:

1. if $(\mathfrak{h}(\mathcal{A}))_1 = 0$, then replace $\mathcal{A}$ by the result of Algorithm 13 given inputs $\mathcal{A}$ and $(\mathfrak{h}(\mathcal{A}))_2$,
2. if $(\mathfrak{h}(\mathcal{A}))_1 = 1$, then
   a) compute the set NewStates by application of Algorithm 14 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $(\mathfrak{h}(\mathcal{A}))_4$,
   b) extract the remaining components:
      i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,
      ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and
      iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

      and then
   c) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,
3. if $(\mathfrak{h}(\mathcal{A}))_1 = 2$, then
   a) compute the set NewStates by application of Algorithm 15 given inputs $\mathcal{A}$ and $(\mathfrak{h}(\mathcal{A}))_3$,
   b) extract the remaining components:
      i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,
      ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

and then

c) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

4. if $(\mathfrak{h}(\mathcal{A}))_1 = 3$, then

a) compute the automaton $\bar{\mathcal{A}} = \bar{\mathfrak{h}}(\mathcal{A}, 0, 1, (\mathfrak{h}(\mathcal{A}))_3)$,

b) compute the set NewStates by application of Algorithm 16 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $\bar{\mathcal{A}}$,

c) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

and then

d) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

5. if $(\mathfrak{h}(\mathcal{A}))_1 = 4$, then

a) compute the automaton $\bar{\mathcal{A}} = \bar{\mathfrak{h}}(\mathcal{A}, 0, 2, (\mathfrak{h}(\mathcal{A}))_3)$,

b) compute the set NewStates by application of Algorithm 17 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $\bar{\mathcal{A}}$,

c) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and

   iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

   and then

  d) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

6. if $(\mathfrak{h}(\mathcal{A}))_1 = 5$, then replace $\mathcal{A}$ by the result of Algorithm 18 given inputs $\mathcal{A}$ and $(\mathfrak{h}(\mathcal{A}))_2$,
7. if $(\mathfrak{h}(\mathcal{A}))_1 = 6$, then
  a) compute the set NewStates by application of Algorithm 19 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $(\mathfrak{h}(\mathcal{A}))_4$,
  b) extract the remaining components:
   i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,
   ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and
   iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

   and then

  c) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

8. if $(\mathfrak{h}(\mathcal{A}))_1 = 7$, then
  a) compute the set NewStates by application of Algorithm 20 given inputs $\mathcal{A}$ and $(\mathfrak{h}(\mathcal{A}))_3$,
  b) extract the remaining components:
   i. extract δ_{new} by application of Algorithm 11 given inputs NewStates and $\Sigma = 2^{\textsc{Var}(\varphi)}$,
   ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NEWSTATES

and then

c) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

9. if $(\mathfrak{h}(\mathcal{A}))_1 = 8$, then

a) compute the automaton $\bar{\mathcal{A}} = \bar{\mathfrak{h}}(\mathcal{A}, 1, 1, (\mathfrak{h}(\mathcal{A}))_3)$,

b) compute the set NEWSTATES by application of Algorithm 16 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $\bar{\mathcal{A}}$,

c) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NEWSTATES and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NEWSTATES as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NEWSTATES

and then

d) replace $\mathcal{A}$ by $(\textsc{NewStates}, \delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}})$,

and

10. if $(\mathfrak{h}(\mathcal{A}))_1 = 9$, then

a) compute the automaton $\bar{\mathcal{A}} = \bar{\mathfrak{h}}(\mathcal{A}, 1, 2, (\mathfrak{h}(\mathcal{A}))_3)$,

b) compute the set NEWSTATES by application of Algorithm 17 given inputs $\mathcal{A}$, $(\mathfrak{h}(\mathcal{A}))_3$ and $\bar{\mathcal{A}}$,

c) extract the remaining components:

i. extract δ_{new} by application of Algorithm 11 given inputs NEWSTATES and $\Sigma = 2^{\textsc{Var}(\varphi)}$,

ii. extract $S_{0,\text{new}}$ from NewStates as described before Theorem 9.2.2 and

iii. extract $\mathcal{F}_{\text{new}}$ by application of Algorithm 12 given input NewStates

and then

d) replace $\mathcal{A}$ by (NewStates, $\delta_{\text{new}}, S_{0,\text{new}}, \mathcal{F}_{\text{new}}$).

The strategy described above is then an implementation of the identification process of Ltl–formulas from sets $\mathcal{E}^+$ and $\mathcal{E}^-$ of positive and negative examples. As soon as the identification process stops we therefore extract the learned formula from the set of initial states of the resulting automaton as described in the proof of Theorem 10.4.5.

The definition of the identification process has shown that Ltl–programs can be learned from sets $\mathcal{E}^+$ and $\mathcal{E}^-$ by application of our algorithms. What remains to be examined is (as in the case of Prolog(+T)–programs) the complexity of the identification task. The following chapter will therefore attack this problem.

11. Identifiability of Ltl–programs

Contents

11.1. General Notations 251

11.2. Upper Bounds for the VC–Dimension of Classes of Ltl–Programs 254

11.2.1. General Ltl–Programs 254

11.2.2. The deterministic Sublogic of Ltl 256

This final chapter will deal with a similar problem as chapter 8.2, namely the problem of estimating the complexity of the identification process of LTL–programs. In chapter 8.2 we have extended some techniques from [11] in order to derive upper bounds for the VC–dimension of several classes of Prolog(+T)–programs. We will proceed similarly for LTL–programs.

11.1. General Notations

We will first identify the objects to be encoded and then show how many bits will be needed in order to encode these objects. It is immediately clear that the propositional symbols from X and the propositional constants `true` and `false` have to be encoded. This yields $|X|+2$ distinct objects. Additionally we will choose the symbols "(", "," and ")" (i.e. the brackets and the comma–symbol), the logical connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and $\neg$ and the temporal operators X, G, F, U and R. Assuming that $X = \{p_1, \dots, p_n\}$ this gives

Symbol	Encoding	Symbol	Encoding
(	bin$(n+3)$	X	bin$(n+11)$
)	bin$(n+4)$	F	bin$(n+12)$
,	bin$(n+5)$	G	bin$(n+13)$
$\wedge$	bin$(n+6)$	U	bin$(n+14)$
$\vee$	bin$(n+7)$	R	bin$(n+15)$
$\rightarrow$	bin$(n+8)$		
$\leftrightarrow$	bin$(n+9)$		
$\neg$	bin$(n+10)$		

Table 11.1.: Encodings for logical connectives from LTL

a total of $n+15$ distinct objects. For the sake of formal clearance we will assume that all formulas are represented in prefix–notation, that is the 2–ary connectives $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$ as well as the 2–ary temporal operators U and R are seen as 2–ary function symbols and the remaining logical and temporal operators are treated as unary function symbols. For example if $\varphi = p_1 \wedge \mathsf{X}p_2\mathsf{U}\mathsf{G}p_3$, then φ is treated as the string $\wedge(p_1, \mathsf{U}(\mathsf{X}(p_2), \mathsf{G}(p_3)))$.

The encoding *cod* which we will use is based on the value n of elements occurring in X. If $p_i \in X$ is any propositional constant, then $cod(p_i) = \text{bin}(i)$ where $\text{bin}(i)$ denotes the binary representation of the integer i. Additionally we define $cod(\texttt{true}) = \text{bin}(n+1)$ and $cod(\texttt{false}) = \text{bin}(n+2)$. The remaining symbols can be encoded in an arbitrary but fixed way. We will choose the encoding from Table 11.1.

The encoding from Table 11.1 has to be changed in the way that the encoding of symbols with strictly less than $|\text{bin}(n+15)|$ symbols are padded with zeros from the left side. This yields equal length for every encoded symbol. For example if $X = \{p_1, p_2, p_3\}$, then $n = 3$ and $n + 15 = 18$. So we have $cod(p_1) = 00001$ and $cod(\mathsf{R}) = 10010$.

Obviously every symbol to be encoded can be represented as a string from $\{0,1\}^*$ of length $\lceil \log_2(n+15) \rceil$. Since the encoding is padded with zeros every formula will be encoded as a string from $\{0,1\}^*$ which has a length which is a multiple of this value.

Having defined the encoding of the symbols used in LTL–formulas it remains to show how composite formulas are encoded. But this is straightforward: Let any LTL–formula

φ be given.

Case 1 if $\varphi = \varphi_1 \oplus \varphi_2$ for $\oplus \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \mathsf{U}, \mathsf{R}\}$, then

$$cod(\varphi) = cod(\oplus)cod(\text{"("})cod(\varphi_1)cod(\text{","})cod(\varphi_2)cod(\text{")"})$$

and

Case 2 if $\varphi = \oplus\psi$ for $\oplus \in \{\neg, \mathsf{X}, \mathsf{G}, \mathsf{F}\}$, then

$$cod(\varphi) = cod(\oplus)cod(\psi)$$

Example 11.1.1

Let $\varphi = \mathsf{G}(p_1 \rightarrow (\mathsf{X}p_2 \rightarrow \neg p_3))$ be given. Due to the assumption from above we will treat φ as $\mathsf{G} \rightarrow (p_1, \rightarrow (\mathsf{X}p_2, \neg p_3))$. We have $X = \{p_1, p_2, p_3\}$ that is $n = 3$ and therefore $n + 15 = 18$. Consequently the symbols are encoded using $\lceil \log_2(18) \rceil = 5$ bits. The encoding of the relevant symbols is therefore: $cod(p_1) = 00001$, $cod(p_2) = 00010$, $cod(p_3) = 00011$, $cod(\text{"("}) = 00100$, $cod(\text{")"}) = 00110$, $cod(\text{","}) = 00101$, $cod(\rightarrow) = 01011$, $cod(\neg) = 01101$, $cod(\mathsf{X}) = 01110$ and $cod(\mathsf{G}) = 10000$. So we have

$$\begin{aligned}
cod(\varphi) &= cod(\mathsf{G}(p_1 \rightarrow (\mathsf{X}p_2 \rightarrow \neg p_3))) \\
&= cod(\mathsf{G}(\rightarrow (p_1, \rightarrow (\mathsf{X}(p_2), \neg(p_3))))) \\
&= cod(\mathsf{G})cod(\text{"("})cod(\rightarrow)\ldots \\
&= \boxed{10000}\boxed{00100}\boxed{01011}\boxed{00100}\boxed{00001}\boxed{00101}\boxed{01011}\boxed{01000}\boxed{01110}\boxed{00100}\ldots \\
&\ldots\boxed{00010}\boxed{00110}\boxed{00101}\boxed{01101}\boxed{00100}\boxed{00011}\boxed{00110}\boxed{00110}\boxed{00110}\ldots \\
&\ldots\boxed{00110}\boxed{00110}
\end{aligned}$$

Some results for the value of VCDIM($\mathcal{C}$) for classes $\mathcal{C}$ consisting of certain propositional logic formulas have been presented before. Early studies by Natarajan (see [124]) merely deal with polynomial time PAC–learnability. Several newer papers also deal with estimations as well as with exact bounds for the VC–dimension of classes of boolean formulas. For example in [135] it is shown that the class $\textsc{Mon}_n$ of monotone boolean formulas consisting of n propositional variables has $\mathrm{VCD}\textsc{im}(\textsc{Mon}_n) = \binom{n}{\lfloor \frac{n}{2} \rfloor}$. Further approaches for the problem of learning propositional formulas which do not use the notion of VC–dimension are for example presented in [141] and [80].

As in chapter 8 we will now proceed by first deriving estimations for an upper bound for the VC–dimension of syntactically unrestricted LTL–formulas and then by studying a more restricted language given by the so–called *deterministic sublogic* $\textsc{Ltl}^{\mathrm{det}}$ of LTL.

11.2. Upper Bounds for the VC–Dimension of Classes of Ltl–Programs

11.2.1. General Ltl–Programs

We will now derive upper bounds for the VC–Dimension of certain classes LTL–programs. For this purpose we define the following: for given values of n, c and t the class $\textsc{Ltl}^{n,c,t}$ denotes the set of all LTL–formulas containing at most n distinct propositional variables, at most c logical connectives and at most t temporal operators.

For the rest of this section assume that n, c and t are fixed nonnegative integers and assume that a formula $\varphi \in \textsc{Ltl}^{n,c,t}$ is chosen. We will define a measuring–function $\|\cdot\| : \textsc{Ltl}^{n,c,t} \to \mathbb{N}$ mapping formulas to integers as follows: $\|\varphi\| = |cod(\varphi)|$, that is $\|\varphi\|$ denotes the number of binary digits in the representation of $cod(\varphi)$.

We will now derive an upper bound for the value $\mathrm{VCD}\textsc{im}(\textsc{Ltl}^{n,c,t})$ by presenting a function $l : \mathbb{N}^3 \to \mathbb{N}$ such that $\|\varphi\| \leq l(n,c,t)$ for any values of n, c and t and every $\varphi \in \textsc{Ltl}^{n,c,t}$. First we have to recall that the propositional constants `true` and `false` and the propositional variables $p_i \in X$ can be encoded using $\lceil \log_2(n+15) \rceil$ bits, that is $\|\varphi\| = \lceil \log_2(n+15) \rceil$ for $\varphi \in X \cup \{\mathtt{true}, \mathtt{false}\}$. Furthermore the connectives and the temporal operators can be encoded with the same number of bits. So we will have to identify the *worst case*, that is the case in which $\varphi \in \textsc{Ltl}^{n,c,t}$ has maximum length. Clearly in this case *exactly* t binary temporal operators have to be involved in φ. Similarly we need c binary connectives in order to reach the maximum length. Each of these operators and connectives is encoded using $\lceil \log_2(n+15) \rceil$ bits. Additionally we need $3\lceil \log_2(n+15) \rceil$ more bits in order to encode the brackets and the commas. Furthermore in this case we have $t+c+1$ subformulas ψ_j without occurrences of any temporal operator or connective, that is $\psi_j \in X \cup \{\texttt{true}, \texttt{false}\}$ for $j = 1, \ldots, t+c+1$. This gives a total number of

$$\begin{aligned} & 4(t+c)\lceil \log_2(n+15) \rceil + (t+c+1)\lceil \log_2(n+15) \rceil \\ = \quad & (5t+5c+1)\lceil \log_2(n+15) \rceil \end{aligned}$$

bits as the maximum value of $\|\varphi\|$ for any $\varphi \in \textsc{Ltl}^{n,c,t}$. But with this number of bits we can encode at most $2^{(5t+5c+1)\lceil \log_2(n+15) \rceil}$ different formulas, that is

$$|\textsc{Ltl}^{n,c,t}| \leq 2^{(5t+5c+1)\lceil \log_2(n+15) \rceil}$$

and so the following theorem can be proved.

Theorem 11.2.1

Let n, c and t be fixed nonnegative integers. Then

$$\textsc{VCDim}\left(\textsc{Ltl}^{n,c,t}\right) = \mathcal{O}\left((5t+5c+1)\log_2(n+15)\right).$$

Proof. Immediately by the above estimation for the size of $\textsc{Ltl}^{n,c,t}$ and Lemma 8.1.1.□

11.2.2. The deterministic Sublogic of Ltl

In the foregoing section we have derived an upper bound for the value of the VC–dimension of structured classes of arbitrary LTL–formulas. Here we will see that a particular subset of LTL, namely the set of all *deterministic* formulas as introduced by [109] can be identified using only marginally more bits in the encoding of formulas.

Roughly speaking the language $\mathrm{LTL}^{\mathrm{det}}$ of *deterministic* LTL–formulas consists of all these elements from LTL in which the usage of the *nondeterministic* operators $\vee$, U and R is guarded by some propositional variable symbol p. Formally $\mathrm{LTL}^{\mathrm{det}}$ is defined as follows:

1. `true` and `false` are in $\mathrm{LTL}^{\mathrm{det}}$,
2. every $p \in X$ is in $\mathrm{LTL}^{\mathrm{det}}$ and
3. for all $\varphi_1, \varphi_2 \in \mathrm{LTL}^{\mathrm{det}}$ and each $p \in X$ $\varphi_1 \wedge \varphi_2$, $\mathsf{X}\varphi_1$, $(p \wedge \varphi_1) \vee (\neg p \vee \varphi_2)$, $(p \wedge \varphi_1)\mathsf{U}(\neg p \vee \varphi_2)$ and $(p \wedge \varphi_1)\mathsf{R}(\neg p \vee \varphi_2)$ are in $\mathrm{LTL}^{\mathrm{det}}$.

As before the set $\mathrm{LTL}^{n,c,t,\mathrm{det}}$ denotes the set of all formulas $\varphi \in \mathrm{LTL}^{\mathrm{det}}$ containing at most n distinct elements from X, at most c connectives and at most t temporal operators.

Since $\mathrm{LTL}^{\mathrm{det}}$–formulas are syntactically more complex than general LTL–formulas, we can ask if this does change the value of $\mathrm{VCDim}\left(\mathrm{LTL}^{n,c,t,\mathrm{det}}\right)$. Below we will see that this is *not* the case.

Of course, the language $\mathrm{LTL}^{\mathrm{det}}$ is less expressive than LTL. But in [109] it is shown that $\mathrm{LTL}^{\mathrm{det}}$–formulas have the property that their negation can be represented by a 1–weak Büchi–automaton (a certain type of Büchi–automaton which is equipped with partial ordering on the set STATES which is compatible with the relation δ) which has a set of states of size linear in the size (that is the length) of φ.

In order to derive an upper bound for the VC–dimension of $\mathrm{LTL}^{\mathrm{det}}$ we will again *rewrite* formulas in prefix notation and change the arity of the nondeterministic symbols $\vee$, U and R to 3 as follows: assume that $\oplus \in \{\vee, \mathsf{U}, \mathsf{R}\}$, $p \in X$ and φ_1, φ_2 are chosen.

Then $\varphi = (p \wedge \varphi_1) \oplus (\neg p \wedge \varphi_2)$ will be rewritten to $\oplus(p, \varphi_1, \varphi_2)$. Using the encoding scheme from the foregoing section we can encode these formulas by setting $\mathit{cod}(\varphi) = \mathit{cod}(\oplus)\mathit{cod}(\text{"("})\mathit{cod}(p)\mathit{cod}(\text{","})\mathit{cod}(\varphi_1)\mathit{cod}(\text{","})\mathit{cod}(\varphi_2)\mathit{cod}(\text{")"})$.

Since the operators F and G do not have to be encoded, we can now encode every symbol occurring in a formula from $\text{LTL}^{n,c,t,\text{det}}$ using at most $\lceil \log_2(n+13) \rceil$ bits. So we clearly have for arbitrary formulas $\varphi \in \text{LTL}^{n,c,t,\text{det}}$:

1. If $\varphi \in X \cup \{\texttt{true}, \texttt{false}\}$, then

$$\|\varphi\| = \lceil \log_2(n+13) \rceil,$$

2. if $\varphi = \mathsf{X}\psi$, then

$$\|\varphi\| = \lceil \log_2(n+13) \rceil + \|\psi\|,$$

3. if $\varphi = \varphi_1 \wedge \varphi_2$, then

$$\|\varphi\| = 4\lceil \log_2(n+13) \rceil + \sum_{i=1}^{2} \|\varphi_i\|$$

and

4. if $\varphi = (p \wedge \varphi_1) \oplus (\neg p \wedge \varphi_2)$ for some $\oplus \in \{\vee, \mathsf{U}, \mathsf{R}\}$, then

$$\|\varphi\| = 6\lceil \log_2(n+13) + \sum_{i=1}^{2} \|\varphi_i\|.$$

The next step is to determine the maximum number of positions in ϕ such that an element from $X \cup \{\texttt{true}, \texttt{false}\}$ occurs at this position. Obviously for $t = 0$ we have at most $c+1$ such positions. In the case that $t > 0$ we can distinguish the following cases:

1. If $\varphi = \mathsf{X}\psi$, then

$$|\{p \in \text{Pos}(\varphi) \mid \varphi|_p \in X \cup \{\texttt{true}, \texttt{false}\}\}|$$

$$= \quad |\{p \in \mathrm{Pos}(\varphi) \mid \psi|_p \in X \cup \{\mathtt{true}, \mathtt{false}\}\}|$$

and

2. if $\varphi = (p \wedge \varphi_1) \oplus (\neg p \wedge \varphi_2)$ for $\oplus \in \{\mathsf{U}, \mathsf{R}\}$, then

$$\begin{aligned} & |\{p \in \mathrm{Pos}(\varphi) \mid \varphi|_p \in X \cup \{\mathtt{true}, \mathtt{false}\}\}| \\ = \quad & 2 + \sum_{i=1}^{2} |\{p \in \mathrm{Pos}(\varphi_i) \mid \psi|_p \in X \cup \{\mathtt{true}, \mathtt{false}\}\}|\,. \end{aligned}$$

Clearly we have the maximum value of such positions for if φ contains t binary temporal operators. It is obvious that in this case we have

$$|\{p \in \mathrm{Pos}(\varphi) \mid \varphi|_p \in X \cup \{\mathtt{true}, \mathtt{false}\}\}| \leq 4t.$$

But since t such binary operators *consume* $3t$ logical connectives there must be $k \geq 0$ such that $c \leq 3t + k$. So additional we have $3t + k + 1$ more such positions. So in total we have $4t + 3t + k + 1 = 7t + k + 1$ such positions. So the elements from $X \cup \{\mathtt{true}, \mathtt{false}\}$ which occur in φ can be encoded using at most

$$(7t + k + 1)\lceil \log_2(n + 13) \rceil$$

bits.

The total value of $\|\varphi\|$ can be estimated using the following two parameters which determine the number of positions with *deterministic* connectives and the number of positions with *nondeterministic* connectives:

$$\begin{aligned} \mathrm{Det}(\varphi) \quad &= \quad |\{p \in \mathrm{Pos}(\varphi) \mid \varphi|_p = \neg\psi \text{ or } \varphi|_p = \varphi_1 \wedge \varphi_2 \text{ or } \varphi|_p = \mathsf{X}\psi\}| \text{ and} \\ \mathrm{NDet}(\varphi) \quad &= \quad |\{p \in \mathrm{Pos}(\varphi) \mid \varphi|_p = (p \wedge \varphi_1) \oplus (\neg p \wedge \varphi_2) \text{ for some } \oplus \in \{\vee, \mathsf{U}, \mathsf{R}\}\}|\,. \end{aligned}$$

Since each of the c connectives in φ is either deterministic or nondeterministic we have

$$\begin{aligned} \|\varphi\| &\leq (7t+k+1)\lceil\log_2(n+13)\rceil + 4\textsc{Det}(\varphi)\lceil\log_2(n+13)\rceil + 6\textsc{NDet}(\varphi)\lceil\log_2(n+13)\rceil \\ &= (7t+k+1+4\textsc{Det}(\varphi)+6\textsc{NDet}(\varphi))\lceil\log_2(n+13)\rceil. \end{aligned}$$

Using the estimations $\textsc{Det}(\varphi) \leq c$ and $\textsc{NDet}(\varphi) \leq t+c$ we have

$$\begin{aligned} \|\varphi\| &\leq (7t+k+t+4c+6(c+t))\lceil\log_2(n+13)\rceil \\ &= (13t+10c+k+1)\lceil\log_2(n+13)\rceil. \end{aligned}$$

Finally we have $k \leq c$ and therefore

$$\begin{aligned} \|\varphi\| &\leq (13t+10c+c+1)\lceil\log_2(n+13)\rceil \\ &= (13t+11c+1)\lceil\log_2(n+13)\rceil. \end{aligned}$$

So any formula $\varphi \in \textsc{Ltl}^{n,c,t,\mathrm{det}}$ can be encoded using at most $(13t+11c+1)\lceil\log_2(n+13)\rceil$ bits and therefore

$$\left|\textsc{Ltl}^{n,c,t,\mathrm{det}}\right| \leq 2^{(13t+11c+1)\lceil\log_2(n+13)\rceil}.$$

So the following theorem is proved.

Theorem 11.2.2
Let n, c and t be fixed nonnegative integers. Then

$$\textsc{VCDim}\left(\textsc{Ltl}^{n,c,t,\mathrm{det}}\right) = \mathcal{O}\left((13t+11c+1)\log_2(n+13)\right).$$

As we have already mentioned in the beginning of this section the value $\textsc{VCDim}\left(\textsc{Ltl}^{n,c,t,\mathrm{det}}\right)$ is not significantly greater than the value of $\textsc{VCDim}\left(\textsc{Ltl}^{n,c,t}\right)$.

The results from the theorems derived in this chapter complete our study on the learnability of LTL–programs from positive and negative examples. We have seen that

Ltl–programs can be identified by applying manipulations to the set of states of certain automata (the representing automata) of actual hypotheses. By application of upward and downward refinement procedures we were able to define an identification process for Ltl–formulas. The following last chapter of this thesis summarizes the results obtained during our studies and points out open problems and directions for future research.

12. Conclusions

This short final chapter will summarize the topics addressed in this thesis and point out some aspects for future research activities.

12.1. Summary of the Topics

We have addressed the problem of *learning* temporal logic programs written in some temporal logic programming language from positive and negative examples. Therefore we have structured the theory into two parts:

First order Inductive Temporal Logic Programming In the case of first order languages we have developed the programming language PROLOG(+T) which is a rule–based, PROLOG–style programming languages which is equipped with the temporal operators X, G, F, U and P. We have described how to prove goals from PROLOG(+T)–programs and have seen how the semantics of PROLOG(+T)–programs can be characterized. While the former topic can be solved using a modified and extended *tableaux* procedure, the latter can be solved by extending the theory of *Herbrand–Interpretations* and *Herbrand–Models* which are a standard tool from the field of first order Logic Programming.

Following the definition and description of the programming language of interest we have seen how specialization and generalization operations can be carried out in order to modify the programs in the case that they contradict the examples.

The last point of the first part dealt with the question of how to analyze the complexity of the learning task. We have seen that the class of PROLOG(+T)–programs can be structured depending on certain syntactic parameters in such a way that it enables the derivation of upper bounds for the VC–Dimension of these classes.

Propositional Inductive Temporal Logic Programming While the techniques developed for first order temporal programming languages face the problem that first order logic in general and first order temporal logic in particular is undecidable, the restriction to propositional temporal logic languages yields decidable satisfiability and implication problems. Therefore we have studied how LTL can be used in order to solve the problem of learning certain temporal logic formulas from positive and negative examples. We have recalled that LTL–formulas can be translated into nondeterministic Büchi–automata and that these automata can be refined in order to fit specifications imposed by the sets of examples. Furthermore we have seen that the generality ordering chosen in the case of first order languages (i.e. the subsumption ordering) cannot be applied in the case of LTL since the objects under consideration are not necessarily clauses. But since propositional temporal programming languages have decidable satisfiability problems we could refine programs with respect to the implication ordering which is finer than subsumption.

As in the case of first order temporal logic we have also studied the complexity of identifying LTL–formulas (resp. LTL–programs) from positive and negative examples by deriving upper bounds for the VC–dimension of certain classes of such formulas.

12.2. Perspectives for Future Research

Three major perspectives can or should be pointed out: Integration of *constraints* (and integration of constraint solving techniques into the theorem proving procedure), the definition of a syntactically less restricted first order language (i.e. relaxing the condition that the objects of PROLOG(+T)–programs are essentially clauses) and studying other, perhaps more expressive propositional temporal languages, such as CTL, CTL*, the μ–calculus or some of the sublogics of the foregoing languages.

Integration of Constraints Constraints are constructs which model relations between certain objects such that the relations have a fixed interpretation. Such relation symbols with fixed interpretation can be the equality symbol $=$ or some comparison symbol such as $\leq$, $<$, $>$ or $\geq$. Of course these symbols are well suited for reasoning about *arithmetic relations* and consequently many constraints are *arithmetic constraints*. The integration of constraints into PROLOG(+T) would make it necessary to extend the theorem proving procedure in order to integrate *constraint solving* resp. *constraint satisfaction* routines (see [84] or [85] for a survey of constraint programming techniques) and techniques for synthesizing logic programs containing constraints (see [144])

Extending the syntax of Prolog(+T) Another perspective for future research might be the extension of PROLOG(+T) in such a way that the program statements need not to be clauses anymore. This would lead us to the full first order linear time temporal logic FOLTL. Dealing with FOLTL objects might cause several problems. On the one hand we can not hope for efficient refinement procedures anymore since the objects under consideration are not necessarily clauses, so subsumption does not make any sense for general FOLTL–formulas. On the other hand the theorem proving task would become much more complicated (see [3] for a description of temporal logic theorem proving in the case of nonclausal languages).

Other propositional temporal languages The restriction to propositional temporal programming languages has (as we have seen in the third part of this thesis) the advantage that satisfiability and implication problems can be decided (in contrast to PROLOG(+T) which contains the full first order predicate logic and which is therefore undecidable). But we have only studied *one* of all possible propositional temporal languages, namely the *linear* time temporal logic LTL. Here the term *linear* means that the language contains only such temporal quantifiers which allow reasoning about *one* possible continuation of the actual point of time under consideration. Branching time Logic (as we will see below) contains other quantifiers which are capable of modeling questions dealing with *all* continuations resp. *some* continuations and which can in these sense be seen as temporal versions of the universal and existential quantifier.

Branching Time Logic As we have already noted above, Branching time temporal logic allows reasoning about more than one continuation of the actual point of time. CTL (another prominent specification language which implements the concept of branching time) is equipped with the quantifiers E and A which model the circumstance that there *exists a computation path* resp. that for *all computation paths* the quantified formula has to hold (see [34] or [58] for a formal description of CTL). Consequently the term CTL stands for *Computation Tree Logic* since the set of possible computations can be seen as trees. CTL–formulas can be translated into Alternating Tree Automata (see [21]) which could also be studied and extended in order to allow refinement operations.

Mixed Logic Combining Linear and Branching Time Temporal Logics one gets the logic CTL* which is more expressive than both CTL and LTL. Formulas from CTL* can be translated into Street tree Automata (see e.g. [59]). This logic could also be studied in order to learn CTL* construct from examples.

Fixpoint Logic From the theoretical point of view the μ–calculus $\mathcal{L}_\mu$ is perhaps the most interesting propositional temporal language. It integrates temporal operators and fixpoint operators (see [96]). It is more expressive than Ltl, Ctl and Ctl* (see [143]).

All these logics are essentially propositional logics and therefore they are decidable for satisfiability. Consequently they should be studied in order to characterize the complexity of identifying concepts from positive and negative examples.

A. Formal Description of the Programming Languages

Contents

A.1. Syntax of Prolog(+T) 267

A.1.1. Terms 268

A.1.2. Atoms and Literals 270

A.1.3. Rules 271

A.1.4. General Prolog(+T)–Objects 271

A.2. Syntax of Ltl 272

For the sake of completeness and in order to make it easier to develop parsers and interpreters for the languages used throughout this thesis we will now give grammars for these languages. The grammars will be presented in an *EBNF*–like syntax, i.e. each grammar will consist of a set of rules with one nonterminal symbol on the left hand side and more or less arbitrary right hand sides. These grammars can be easily converted into a form which is suitable for tools generating compilers (e.g. `bison`, see [68]).

A.1. Syntax of Prolog(+T)

By definition we have several different types of objects which have to be generated by the grammar. These are terms, atoms and literals and rules (including facts as special cases).

Consequently we have to present rules which are capable to define all these objects.

A.1.1. Terms

Terms have been defined to be constructs of the following form:

- Variable Terms, i.e. constructs such as $t = x$ for symbols $x \in \mathcal{X}$,
- Strings representing Integers from $\mathbb{Z}$,
- Function Terms, i.e. constructs such as $t = f(t_1, \ldots, t_n)$ or
- List Terms, that is $t = []$ or $t = [t_1, \ldots, t_n]$ or $t = [\bar{t}|t_1, \ldots, t_n]$.

Consequently the first production step for the generation of a term must chose which type of term has to be created.

Term	::=	*Variable–symbol* | *Function–Term* | *List–Term* | *Number*

Variable–Terms and Integers

Depending on the type of term to be generated we have to give rules for generating each such type. Therefore we make the following convention:

- a variable symbol has to start with an upper case letter and
- any other symbol (i.e. a function or a predicate symbol) has to start with a lower case letter.

The simplest objects to be created are variable terms and integers.

Variable–symbol	::=	'_' | *Upper–Case–Letter* |
		Upper–Case–Letter Variable–Suffix
Upper–Case–Letter	::=	'A' |'B' |'C' |'D' |'E' |'F' |'G' |'H' |'I' |'J' |'K' |'L' |'M' |
		'N' |'O' |'P' |'Q' |'R' |'S' |'T' |'U' |'V' |'W' |'X' |'Y' |'Z'

Lower–Case–Letter	::=	‘a’ |‘b’ |‘c’ |‘d’ |‘e’ |‘f’ |‘g’ |‘h’ |‘i’ |‘j’ |‘k’ |‘l’ |‘m’ |
		‘n’ |‘o’ |‘p’ |‘q’ |‘r’ |‘s’ |‘t’ |‘u’ |‘v’ |‘w’ |‘x’ |‘y’ |‘z’
Letter	::=	*Upper–Case–Letter* | *Lower–Case–Letter*
Digit	::=	‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ |‘8’ | ‘9’
Special–Symbol	::=	*Letter* | *Digit* | ‘_’
Variable–Suffix	::=	*Special–Symbol* | *Special–Symbol Special–Symbol*
Number	::=	*Positive–Number* | *Negative–Number*
Positive–Number	::=	*Digit* | *Digit Positive–Number*
Negative–Number	::=	‘-’ *Positive–Number*

Function–Terms

The next more complicated constructs are function terms, i.e. terms of the form $t = f(t_1, \ldots, t_n)$ with $n = \alpha(f)$ and terms $t_1, \ldots, t_n$ or $t = f$ for constant symbols f (i.e. such symbols f with $\alpha(f) = 0$).

Function–Term	::=	*Function–Name* |
		Function–Name Left–Delimiter
		Argument–List Right–Delimiter
Left–Delimiter	::=	‘(’
Right–Delimiter	::=	‘)’
Function–Name	::=	*Lower–Case–Letter* | *Lower–Case–Letter Function–Suffix*
Function–Suffix	::=	*Variable–Suffix*
Argument–List	::=	*Term* | *Term* ‘,’ *Argument–List*

List–Terms

List terms are the most complicated terms to generate since they might have several different forms. A term t is a list term if it is $t = []$ (i.e. t is the empty list) or if t consists of an enumeration of elements (i.e. $t = [t_1, \ldots, t_n]$ for terms $t_1, \ldots, t_n$) or if t is given by its head and its tail. In this case we can distinguish the cases that $t = [\bar{t}|t_1, \ldots, t_n]$ and $t = [\bar{t}|t']$ where $\bar{t}$ denotes the head and $t_1, \ldots, t_n$ is an enumeration of the elements in the tail or t' is a term representing the whole tail of t.

List–Term	::=	*Left–List–Delimiter Right–List–Delimiter* |
		Left–List–Delimiter List–Arguments
		Right–List–Delimiter
Left–List–Delimiter	::=	‘[’
Right–List–Delimiter	::=	‘]’
List–Arguments	::=	*Argument–List* |
		Term ‘|’ *Argument–List* |
		Term ‘|’ *Term*

A.1.2. Atoms and Literals

Atoms and literals are either temporal or nontemporal ones. Nontemporal atoms and literals are then either one of the constants `true` and `false` or built using predicate symbols. In this case they have the form $p(t_1, \ldots, t_n)$ or `not`$(p(t_1, \ldots, t_n))$.

Atom	::=	‘`true`’ | ‘`false`’ | *Relational–Atom*
Relational–Atom	::=	*Predicate–Name* |
		Predicate–Name Left–Delimiter
		Argument–List Right–Delimiter
Predicate–Name	::=	*Lower–Case–Letter* | *Lower–Case–Letter Predicate–Suffix*
Predicate–Suffix	::=	*Variable–Suffix*

Literal	::=	*Atom* | ‘`not`’ *Left–Delimiter Literal Right–Delimiter*

In order to integrate the temporal operators we generalize the concept of literals to *general literals.* General literals are then divided into two classes, namely temporal literals and nontemporal literals.

General–Literal	::=	*Literal* | *Temporal–Literal* |
		‘`not`’ *Left–Delimiter General–Literal*
		Right–Delimiter
Temporal–Literal	::=	*Unary–Temporal–Literal* |
		Binary–Temporal–Literal
Unary–Temporal–Literal	::=	*Unary–Temporal–Connective General–Literal*
Binary–Temporal–Literal	::=	*General–Literal*
		Binary–Temporal–Connective General–Literal
Unary–Temporal–Connective	::=	‘`X`’ | ‘`F`’ | ‘`G`’
Binary–Temporal–Connective	::=	‘`U`’ | ‘`P`’

A.1.3. Rules

Rules are created using the productions for literals. Each rule is either a fact or a *definite rule*, that is a rule with a nonempty tail.

Rule	::=	*Fact* | *Definite–Rule*
Fact	::=	*General–Literal End–Delimiter*
Definite–Rule	::=	*General–Literal Implication–Symbol*
		List–Of–Literals End–Delimiter
End–Delimiter	::=	‘.’
Implication–Symbol	::=	‘:–’
List–Of–Literals	::=	*General–Literal* | *General–Literal* ‘,’ *List–Of–Literals*

A.1.4. General Prolog(+T)–Objects

Since every Prolog(+T)–object is either a term or a formula, we add a rule produce these two types of objects. Each formula is then given as a literal (a general literal) or a rule.

Prolog(+T)–Object	::=	*Term* | *Formula*
Formula	::=	*General–Literal* | *Rule*

These rules complete the syntax of PROLOG(+T). In order to be well formed an object needs

- to be parsed and then
- to be checked if the symbols used in the object are compatible with the constraints given by the signature.

A.2. Syntax of Ltl

The syntax of LTL is very simple, since there are nearly no constraints on the form of a formula. Consequently a grammar which generates the set of all LTL–formulas can be extracted directly from the definition of the language LTL (see page 20).

Formulas from LTL are either *atomic* formulas (i.e. the constants `true` and `false` or proposition symbols) or composite formulas. Composite formulas are built using unary or binary connectives which can be either propositional or temporal ones.

LTL–Formula	::=	*LTL–Atom* | *LTL–Composite–Formula*
LTL–Atom	::=	‘`true`’ | ‘`false`’ | *LTL–Proposition–Symbol*
LTL–Composite–Formula	::=	*LTL–Unary* | *LTL–Binary*

For the sake of simplicity we will introduce names for the symbols used in the rules to generate the formulas, in particular we will introduce names for the brackets and the connectives.

Left–Delimiter	::=	‘(’
Right–Delimiter	::=	‘)’
LTL–Unary–Propositional–Connective	::=	‘!’

LTL–Unary–Temporal–Connective	::=	‘X’ | ‘G’ | ‘F’
LTL–Binary–Propositional–Connective	::=	‘+’ | ‘∗’ | ‘→’ | ‘↔’
LTL–Binary–Temporal–Connective	::=	‘U’ | ‘R’

Proposition symbols are used in order to build atomic formulas of LTL. They have to start with a letter (no matter if it is an upper case letter or a lower case letter) followed by a (possibly empty) string of arbitrary symbols. Such strings are generated using the rule below which has the symbol *symbol–suffix* on its left hand side.

LTL–Proposition–Symbol	::=	*Nondigit* | *Nondigit Symbol–Suffix*
Nondigit	::=	‘a’ |‘b’ |‘c’ |‘d’ |‘e’ |‘f’ |‘g’ |‘h’ |‘i’ |‘j’ |‘k’ |‘l’ |‘m’ |‘n’ |
		‘o’ |‘p’ |‘q’ |‘r’ |‘s’ |‘t’ |‘u’ |‘v’ |‘w’ |‘x’ |‘y’ |‘z’ |
		‘A’ |‘B’ |‘C’ |‘D’ |‘E’ |‘F’ |‘G’ |‘H’ |‘I’ |‘J’ |‘K’ |‘L’ |
		‘M’ | ‘N’ |‘O’ |‘P’ |‘Q’ |‘R’ |‘S’ |‘T’ |‘U’ |‘V’ |‘W’ |
		‘X’ |‘Y’ |‘Z’
Digit	::=	‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
Symbol–Suffix	::=	‘’ | *Digit Symbol–Suffix* | *Nondigit Symbol–Suffix* |
		‘_’ *Symbol–Suffix*

As we have already mentioned above, composite LTL–formulas can be built up using unary and binary connectives which can be either propositional ones or temporal ones. The rules for building such formulas are given below. First we have to present two rules for building general formulas using binary and unary connectives.

LTL–Unary	::=	*LTL–Negation* | *LTL–Unary–Temporal–Formula*
LTL–Binary	::=	*LTL–Binary–Propositional–Formula* |
		LTL–Binary–Temporal–Formula

Having chosen the arity of the connective rules for generating the formulas using connectives with the chosen arity have to be applied.

LTL–Negation	::=	*LTL–Unary–Propositional–Connective*
		LTL–Formula
LTL–Unary–Temporal–Formula	::=	*LTL–Unary–Temporal–Symbol*
		LTL–Formula
LTL–Binary–Propositional–Formula	::=	*Left–Delimiter LTL–Formula*
		LTL–Binary–Propositional–Connective
		LTL–Formula Right–Delimiter
LTL–Binary–Temporal–Formula	::=	*Left–Delimiter LTL–Formula*
		LTL–Binary–Temporal–Connective
		LTL–Formula Right–Delimiter

Since there are no further restrictions on the syntax of LTL–formulas the grammar for LTL is now complete.

List of Figures

3.1. SLD–refutation 40

5.1. Rewrite System for computing reduced literals 75

5.2. Expansion Rules 95

5.3. Saturation Rules 97

6.1. Unification vs. Anti–unification 118

6.2. TREE(q(a)Up(b)) 132

6.3. TREE(q(c)Up(a)) 132

6.4. TREE(q(c)UXp(a)) 132

7.1. Growth rate of refined rules with and without elimination of variants . . . 163

7.2. Growth Rate by adding Premises only 167

8.1. Number of examples given fixed values for c, t, l, o, p and a with ε ranging from 0 to $\frac{1}{2}$ and δ ranging from 0 to 1. 182

List of Tables

7.2. Set of downward refinements of $\varphi = \mathsf{GXF}\mathtt{p}(\mathtt{X})\mathsf{U}\mathtt{p}(\mathtt{g}(\mathtt{f}(\mathtt{X}),\mathtt{Y}))$ 143

7.3. Set of upward refinements for $\varphi = \mathsf{FG}\mathtt{p}(\mathtt{X}_1,\mathtt{f}(\mathtt{a},\mathtt{f}(\mathtt{X}_1,\mathtt{f}(\mathtt{a},\mathtt{f}(\mathtt{a},\mathtt{a})))),\mathtt{f}(\mathtt{X}_1,\mathtt{X}_2))$ 148

7.5. Set of downward refinements of $C = \mathsf{G}\mathtt{p}(\mathtt{X}_1) :\!- \mathsf{F}\mathtt{q}(\mathtt{X}_1,\mathtt{f}(\mathtt{X}_2)),\mathtt{q}(\mathtt{a},\mathtt{a})\mathsf{U}\mathtt{p}(\mathtt{a})$. (part 1) 153

7.7. Set of downward refinements of $C = \mathsf{G}\mathtt{p}(\mathtt{X}_1) :\!- \mathsf{F}\mathtt{q}(\mathtt{X}_1,\mathtt{f}(\mathtt{X}_2)),\mathtt{q}(\mathtt{a},\mathtt{a})\mathsf{U}\mathtt{p}(\mathtt{a})$. (part 1) 154

7.9. Set of downward refinements of $C = \mathsf{G}\mathtt{p}(\mathtt{X}_1) :\!- \mathsf{F}\mathtt{q}(\mathtt{X}_1,\mathtt{f}(\mathtt{X}_2)),\mathtt{q}(\mathtt{a},\mathtt{a})\mathsf{U}\mathtt{p}(\mathtt{a})$. (part 3) 156

11.1. Encodings for logical connectives from LTL 252

List of Algorithms

1. Tableaux algorithm for ground goals 103

2. Anti–unification for terms 118
3. Greatest Specialization of FoLtl–literals 123
4. Least Generalization of FoLtl–literals 124
5. Greatest Specialization of Prolog(+T)–rules 127
6. Least Generalization of Prolog(+T)–rules 138

7. Downward–Refinement of Prolog(+T)–literals 144
8. Upward–Refinement of Prolog(+T)–literals 149
9. Downward–Refinement of Prolog(+T)–rules 155
10. Upward–Refinement of Prolog(+T)–rules 158

11. Extraction of δ from Σ and States 207
12. Extracting the Acceptance Component $\mathcal{F}$ from States 209

13. Propositional Upward Refinement 222
14. Temporal Upward Refinement: Constructing new States for $\mathsf{F} \mapsto \mathsf{X}^i$ 227
15. Temporal Upward Refinement: Constructing new States for $\mathsf{X}^i \mapsto \mathsf{G}$ 228
16. Temporal Upward Refinement: Constructing new States for Replacement of Eventualities (replacing the first component) 229

17. Temporal Upward Refinement: Constructing new States for Replacement of Eventualities (replacing the second component) 230
18. Propositional Downward Refinement 232
19. Temporal Downward Refinement: Constructing new States for $\mathsf{G} \mapsto \mathsf{X}^i$ . . 235
20. Temporal Downward Refinement: Constructing new States for $\mathsf{X}^i \mapsto \mathsf{F}$ . . 236

Bibliography

[1] M. Abadi and Z. Manna, *A Timely Resolution*, Symposium on Logic in Computer Science, IEEE, 1986, pp. 176–186.

[2] ______, *Temporal Logic Programming*, Journal of Symbolic Computation **8** (1988), 277–295.

[3] ______, *Nonclausal Deduction in First–Order Temporal Logic*, Journal of the ACM **37** (1990), no. 2, 279–317.

[4] S. Akama, *Tableaux for logic programming with strong negation*, Automated Reasoning with Analytic Tableaux and Related Methods, Springer Verlag, 1997, Lecture Notes in Computer Science 1227, pp. 31–42.

[5] H. Andreka and I. Nemeti, *The Generalized Completeness of Horn Predicate Logic as a Programming Language*, Acta Cybernetica **4** (1978), 3–10.

[6] T. Aoyagi, M. Fujita, and T. Moto-Oka, *Temporal Logic Programming Language Tokio: Programming in Tokio*, Proceedings of the 4th Conference on Logic Programming, Springer Verlag, 1985, Lecture Notes in Computer Science 221, pp. 128–137.

[7] K.R. Apt, *Logic programming*, Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), Elsevier, 1990, pp. 493–574.

[8] ______, *From Logic Programming to Prolog*, Prentice Hall, 1997.

[9] K.R. Apt and A. Pellegrini, *On the occur-check-free PROLOG programs*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 3, 687–726.

[10] K.R. Apt and M.H. van Emden, *Contributions to the Theory of Logic Programming*, Journal of the ACM **29** (1984), no. 3, 841–862.

[11] M. Arias and R. Khardon, *Complexity Parameters for First Order Classes*, Tech. Report 2004–6, Tufts University, July 2004.

[12] J. Avenhaus, *Reduktionssysteme*, Springer Verlag, 1995, (in German).

[13] F. Baader and T. Nipkow, *Term Rewriting and All That*, Cambridge University Press, 1999.

[14] M. Baaz, A. Leitsch, and R. Zach, *Completeness of a First-order Temporal Logic with Time-Gaps*, Theoretical Computer Science **160** (1996), 241–270.

[15] L. Badea, *A Refinement Operator For Theories*, Proceedings of the 15th International Conference on Inductive Logic Programming, Springer Verlag, 2001, Lecture Notes in Computer Science 2175.

[16] M. Bain and A. Srinivasan, *Inductive Logic Programming with large-scale unstructured Data*, Machine Intelligence **14** (1996), 233–267.

[17] J. Barnat, L. Brim, and J. Chaloupka, *Parallel Breadt–first Search* LTL *Model Checking*, Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE '03), IEEE, 2003, pp. 106–115.

[18] D.W. Barnes and J.M. Mack, *An algebraic Introduction to Mathematical Logic*, Springer Verlag, 1975.

[19] P. Baumgartner and U. Furbach, *Calculi for Disjunctive Logic Programming*, Tech. Report 13–96, Universität Koblenz–Landau, 1995.

[20] M. Ben-Ari, *Mathematical Logic for Computer Science*, second ed., Springer Verlag, 2001.

[21] O. Bernholtz, M. Vardi, and P. Wolper, *An Automata–Theoretic Approach to Branching–Time Model Checking*, Proceedings of the 6th International Conference on Computer Aided Verification, Springer Verlag, 1994, Lecture Notes in Computer Science 818, pp. 142–155.

[22] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth, *Learnability and the Vapnik-Chervonenkis-Dimension*, Journal of the ACM **36** (1989), no. 4, 929–965.

[23] G. Boole, *An Investigation of the Laws on which are founded the Mathematical Theories of Logic and Probabilities*, Dover, New York, 1958.

[24] P.S. Braddock, D.E. hu, I.J. Stratford, A.L. Harris, and R. Bicknell, *A structure–activity analysis of antagonism of the growth factor and angiogenic activity of basic fibriblast growth factor by Suramin and related polyanions*, British Journal of Cancer **69** (1994), no. 5, 890–898.

[25] I. Bratko, S. Muggleton, and A. Varšek, *Learning qualitative models of dynamic systems*, Inductive Logic Programming (S. Muggleton, ed.), 1991, pp. 207–224.

[26] J.R. Büchi, *Weak second–order arithmetic and finite Automata*, Zeitschrift für mathematische Logik und Grundlagen der Mathematik **6** (1960), 60–92.

[27] C. Castellini, *Automated Reasoning in Quantified Modal and Temporal Logics*, Ph.D. thesis, School of Informatics, University of Edinburgh, 2005.

[28] S. Cerrito, M. Cialdea Mayer, and S. Praud, *A Tableau Calculus for First Order Linear Temporal Logic over Bounded Time Structures*, Tech. Report LRI n. 1207, Dipartimento di Informatica e Automazione Università degli studi Roma Tre, Dipartimento di Informatica e Automazione, 1999.

[29] ______, *First Order Linear Temporal Logic over Finite Time Structures*, Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99), Springer Verlag, 1999, Lecture Notes in Computer Science 1705, pp. 62–76.

[30] ______, *First Order Linear Temporal Logic over Finite Time Structures is not semi-decidable*, Tech. Report LRI n. 1208, Dipartimento di Informatica e Automazione Università degli studi Roma Tre, Dipartimento di Informatica e Automazione, 1999.

[31] C.L. Chang and C.T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, third ed., Academic Press, 1990.

[32] A. Church, *A Note on the Entscheidungsproblem*, Journal of Symbolic Logic **1** (1936), 40–41.

[33] P. Clark and R. Boswell, *Rule induction with cn2: some recent improvements*, Proceedings of the fifth European Working Session on Learning, Springer Verlag, 1991, pp. 151–163.

[34] E.M. Clarke and E.A. Emerson, *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*, Logic of Programs, 1981, pp. 52–71.

[35] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, 1999.

[36] W. Cohen, *PAC-learning Recursive Logic Programs: Efficient Algorithms*, Journal of Artificial Intelligence Research **2** (1995), 501–539.

[37] ______, *PAC-learning Recursive Logic Programs: Negative Results*, Journal of Artificial Intelligence Research **2** (1995), 541–573.

[38] S. Cook, *The Complexity of Theorem Proving Procedures*, Proc. 3rd ACM Symp. on Theory of Computing, ACM Press, 1971, pp. 151–158.

[39] M. Daniele, F. Giunchiglia, and M.Y. Vardi, *Improved Automata Generation for Linear Temporal Logic*, CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification, Springer Verlag, 1999, Lecture Notes in Computer Science 1855, pp. 249–260.

[40] A.K. Debnath, R.L. Lopez de Compadre, G. Debnath, A.J. Shusterman, and C. Hansch, *Structure–activity relationship aromatic and heteroatomatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity*, Journal of Medicinical Chemistry **34** (1991), no. 2, 786–797.

[41] A. Degtyarev and M. Fisher, *Towards First–Order Temporal Resolution*, Proceedings of KI, the 24th German Conference on Artificial Intelligence, Springer Verlag, 2001, Lecture Notes in Computer Science 2174, pp. 18–32.

[42] A. Degtyarev, M. Fisher, and B. Konev, *Exploring the monodic fragment of first-order temporal logic using clausal temporal resolution*, Tech. Report ULCS-03-012, University of Liverpool, Department of Computer Science, 2003.

[43] ______, *Monodic Temporal Resolution*, Proceedings of the 19th Conference on Automated Deduction, CADE–19, Springer Verlag, 2003, Lecture Notes in Computer Science 2741, pp. 397–411.

[44] ______, *Monodic Temporal Resolution*, ACM Transactions on Computational Logic **7** (2006), no. 1, 108–150.

[45] L.P. Devroye, *Automatic Pattern Recognition: A Study of the Probability of Error*, IEEE Transactions on Pattern Analysis and Machine Intelligence **10** (1988), no. 4, 530–543.

[46] L.P. Devroye and T.J. Wagner, *A distribution–free Performance Bound in Error Estimation*, IEEE Transactions on Information Theory **22** (1976), no. 5, 586–587.

[47] C. Dixon, *Search Strategies for Resolution in Temporal Logics*, Proceedings of the International Conference on Automated Deduction (CADE-13), Springer Verlag, 1996, Lecture Notes in Computer Science 1104, pp. 673–687.

[48] ______, *Temporal Resolution: Removing Irrelevant Information*, Proceedings of the Fourth International Workshop on Temporal Reasoning (TIME'97), IEEE Press, 1997, pp. 4–11.

[49] C. Dixon and M. Fisher, *The Set of Support Strategy in Temporal Resolution*, Proceedings of the Fifth International Workshop on Temporal Reasoning (TIME'98), IEEE Press, 1998, pp. 113–120.

[50] B. Dolsak, *Constructing finite element meshes usign artificial intelligence methods*, Master's thesis, University of Maribor, 1991.

[51] B. Dolsak, A. Jezernik, and I. Bratko, *A knowledge base for finite element mesh design*, Proceedings of the sixth ISSEK Workshop, 1992.

[52] B. Dolsak and S. Muggleton, *The application of Inductive Logic Programming to finite element mesh design*, Inductive Logic Programming (S. Muggleton, ed.), Academic Press, London, 1992.

[53] B. Dutertre, *Complete Proof Systems for First Order Interval Temporal Logic*, Logic in Computer Science, 1995, pp. 36–43.

[54] S. Dzeroski and I. Bratko, *Handling noise in inductive logic programming*, Proceedings of the Second International Workshop on Inductive Logic Programming, 1992.

[55] S. Dzeroski and B. Dolsak, *Comparison of ilp systems on the problem of finite element mesh design*, Proceedings of the sixth ISSEK Workshop, 1992.

[56] E.A. Emerson, *Temporal and modal logic*, Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), Elsevier, 1990, pp. 995–1072.

[57] ———, *Automated Temporal Reasoning for Reactive Systems*, Logic for Concurrency: Structure versus Automata, Springer Verlag, 1996, pp. 41–101.

[58] E.A. Emerson and E.M. Clarke, *Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*, Science of Computer Programming **2** (1982), no. 3, 241–266.

[59] E.A. Emerson and A.P. Sistla, *Deciding Full Branching Time Logic*, Information and Control **61** (1984), no. 3, 175–201.

[60] J. Esparza and K. Heljanko, *A New Unfolding Approach to* LTL *Model Checking*, Proceedings of the 27th International Colloquium on Automata, Languages and Programming, Springer Verlag, 2000, Lecture Notes in Computer Science 1853, pp. 475–486.

[61] A. Felty, *Temporal Logic Theorem Proving and its Application to the Feature Interaction Problem*, Tech. Report DII 14/01, University of Siena, 2001, in E. Giunchiglia and F. Massacci (ed.): Issues in the Design and Experimental Evaluation of Systems for Modal and Temporal Logics.

[62] A. Felty and L. Thery, *Interactive Theorem Proving with Temporal Logic*, Journal of Symbolic Computation **23** (1997), no. 4, 367–397.

[63] P. Fischer, *Algorithmisches Lernen*, Teubner Verlag, 1999, (in German).

[64] M. Fisher, *A Normal Form for Temporal Logics and its Applications in Theorem-Proving and Execution*, Journal of Logic and Computation **7** (1997), no. 4, 429–456.

[65] M. Fisher, C. Dixon, and M. Peim, *Clausal Temporal Resolution*, ACM Transactions on Computational Logic **2** (2001), no. 1, 12–56.

[66] M. Fitting, *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, 1990.

[67] ———, *Tableaux for Logic Programming*, Journal of Automated Reasoning **13** (1994), no. 2, 175–188.

[68] Free Software Foundation, *Bison 2.3, the yacc–compatible parser generator, manual*, `www.gnu.org/software/bison/manual/pdf/bison.pdf`.

[69] G. Frege, *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*, in [165], Halle, 1879.

[70] U. Furbach, P. Baumgartner, and F. Stolzenburg, *Model Elimination, Logic Programming and Computing Answers*, Tech. Report 1–95, Universität Koblenz–Landau, 1995.

[71] M. R. Garey and D. S. Johnson, *Computers and Intractability – A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[72] P. Gastin and D. Oddoux, *Fast LTL to Büchi Automata Translation*, Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01), Springer Verlag, 2001, Lecture Notes in Computer Science 2102, pp. 53–65.

[73] M. Gelfond and V. Lifschitz, *The Stable Model Semantics for Logic Programming*, Proceedings of the Fifth International Conference on Logic Programming (Cambridge, Massachusetts) (R.A. Kowalski and K. Bowen, eds.), The MIT Press, 1988, pp. 1070–1080.

[74] R. Gentilini, C. Piazza, and A. Policriti, *Computing Strongly Connected Components in a Linear Number of Symbolic Steps*, Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, Maryland), Society for Industrial and Applied Mathematics, 2003, pp. 573–582.

[75] G. Gentzen, *Untersuchungen über das logische Schließen*, Mathematische Zeitschrift **39** (1935), 176–210 and 405–431.

[76] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper, *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, Protocol Specification Testing and Verification (Warsaw, Poland), Chapman & Hall, 1995, pp. 3–18.

[77] G. Gottlob, *Subsumption and Implication*, Information Processing Letters **24** (1987), no. 2, 109–111.

[78] G. Gottlob and C.G. Fermüller, *Removing Redundancy from a Clause*, Artificial Intelligence **61** (1993), no. 2, 263–289.

[79] N. Helft, *Induction as nonmonotonic inference*, Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann, 1989, pp. 149–156.

[80] A. Hernandez-Aguirre, B.P. Buckles, and C.A. Coello Coello, *On Learning $kDNF_n^s$ Boolean Formulas*, Proceedings of the The 3rd NASA/DoD Workshop on Evolvable Hardware, IEEE Computer Society, 2001, pp. 240–248.

[81] J. Hintikka, *Knowledge and Belief*, Cornell University Press: Ithaca, NY, 1962.

[82] I. Hodkinson, F. Wolter, and M. Zakharyaschev, *Decidable fragments of first-order temporal logics*, Annals of Pure and Applied Logic **106** (2000), 85–134.

[83] G. Huet, *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, Journal of the ACM **27** (1980), 797–821.

[84] Joxan Jaffar and Michael J. Maher, *Constraint Logic Programming: A Survey*, Journal of Logic Programming **19/20** (1994), 503–581.

[85] J.-P. Jouannaud and R. Treinen, *Constraints and Constraint Solving: An Introduction*, Constraints in Computational Logics: Theory and Applications (H. Comon, C. Marche, and R. Treinen, eds.), Springer Verlag, 1999, pp. 1–46.

[86] A. Karmath and R.D. King, *An automated ILP Server in the Field of Bioinformatics*, Proceedings of the 15th International Conference on Inductive Logic Programming, Springer Verlag, 2001, Lecture Notes in Computer Science 2175, pp. 91–103.

[87] R.D. King, S. Muggleton, and M.J.E. Sternberg, *Drug Design by Machine Learning: The use of Inductive Logic programming to model the structure–activity relationships of Trimethoprom analogues binding to Dihydrofolate reductase*, Proceedings of the National Academy of Sciences **89** (1992), no. 23, 11322–11326.

[88] R.D. King, A. Srinivasan, and M.J.E. Sternberg, *Relating chemical activity to structure: An examination of ILP successes*, New Generation Computing, Special issue on Inductive Logic Programming **13** (1995), no. 3/4, 411–434.

[89] S.C. Kleene, *Introduction to Metamathematics*, 7th ed., North Holland, 1971.

[90] D. E. Knuth and P. B. Bendix, *Simple Word Problems in Universal Algebra*, Computational Problems in Abstract Algebra (J. Leech, ed.), Pergamon Press, 1970, pp. 263–297.

[91] B. Konev, A. Degtyarev, C. Dixon, M. Fisher, and U. Hustadt, *Mechanising first-order temporal resolution*, Information and Computation **199** (2003), no. 1–2, 55–86.

[92] S. Kono, T. Aoyagi, M. Fujita, and H. Tanaka, *Implementation of Temporal Programming Language Tokio*, Proceedings of the 4th Conference on Logic Programming, Springer Verlag, 1985, Lecture Notes in Computer Science 221, pp. 138–147.

[93] R.A. Kowalski, *Predicate Logic as a Programming Language*, Information Processing '74 (J.L. Rosenfeldt, ed.), North Holland, 1974, pp. 569–574.

[94] ———, *Algorithm=Logic+Control*, Communications of the ACM **22** (1979), no. 7, 424–436.

[95] R.A. Kowalski and D. Kuehner, *Linear Resolution with Selection Function*, Artificial Intelligence **2** (1971), no. 3,4, 227–260.

[96] D. Kozen, *Results on the Propositional mu-Calculus*, Theoretical Computer Science **27** (1983), 333–354.

[97] S. Kripke, *Semantical analysis of modal logic*, Zeitschrift für Mathematische Logik und Grundlagen der Mathematik **9** (1963), 67–96.

[98] P.D. Laird, *Learning from good and bad data*, Kluwer Academic Publishers, 1988.

[99] N. Lavrac and S. Dzeroski, *Inductive Logic Programming: Techniques and Applications* , Ellis Horwood, 1994.

[100] N. Lavrac, S. Dzeroski, V. Pirnat, and V. Krizman, *Learning rules for early diagnosis of rheumatic diseases*, Proceedings of the 3rd Scandinavian Conference on Artificial Intelligence, IOS Press, Amsterdam, 1992, pp. 138–149.

[101] N. Lavrac, I. Kononenko, E. Keravnou, M. Kukar, and B. Zupan, *Intelligent Data Analysis for Medical Diagnosis: Using Machine Learning and Temporal Abstraction*, AI Communications **11** (1998), no. 3–4, 191–218.

[102] O. Lichtenstein and A. Pnueli, *Checking that finite state concurrent Programs satisfy their linear Specifications*, Proceedings of the 12th ACM Symposium on Principles of Programming Languages, ACM, 1985, pp. 97–107.

[103] O. Lichtenstein, A. Pnueli, and L. Zuck, *The Glory of the Past*, Logics of Programs (R. Parikh, ed.), Springer Verlag, 1985, Lecture Notes in Computer Science 193, pp. 196–218.

[104] H. Liebig and S. Thome, *Logischer Entwurf digitaler Systeme*, third ed., Springer Verlag, 1996, (in German).

[105] J.W. Lloyd, *Foundations of Logic Programming*, Springer Verlag, 1987.

[106] M. Lothaire, *Algebraic Combinatorics on Words*, Cambridge University Press, 2002.

[107] D.W. Loveland, *Mechanical Theorem Proving by Model Elimination*, Journal of the ACM **15** (1968), no. 2, 236–251.

[108] ______, *A simplified Format for the Model Elimination Theorem-Proving Procedure*, Journal of the ACM **16** (1969), no. 3, 349–363.

[109] M. Maidl, *The Common Fragment of* CTL *and* LTL, Proceedings of the 41th Annual Symposium on Foundations of Computer Science (FOCS '00), 2000, pp. 643–652.

[110] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer Verlag, 1992.

[111] Z. Manna and P. Wolper, *Synthesis of communicating Processes from Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems **6** (1984), no. 1, 68–93.

[112] K. Marriott and H. Sondergaard, *On prolog and the occur check problem*, ACM SIGPLAN Notices **24** (1989), no. 5, 76–82.

[113] A. Martelli and U. Montanari, *An Efficient Unification Algorithm*, ACM Transactions on Programming Languages and Systems **4** (1982), no. 2, 258–282.

[114] W. May and P.H. Schmitt, *A Tableau Calculus for First–Order Branching Time Logic*, Proceedings of the International Conference on Formal and Applied Practical Reasoning, FAPR-96, Springer Verlag, 1996, Lecture Notes in Computer Science 1085, pp. 399–413.

[115] R. Mooney, P. Melville, L. Tang, J. Shavlik, I. Dutra, D. Page, and V. Santos Costa, *Relational Data Mining with Inductive Logic Programming for Link Discovery*, Proceedings of the National Science Foundation Workshop on Next Generation Data Mining, 2002.

[116] ______, *Relational Data Mining with Inductive Logic Programming for Link Discovery*, Data Mining: Next Generation Challenges and Future Directions (H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha, eds.), AAAI Press, 2004, pp. 239–254.

[117] B. Moszkowski, *Executing Temporal Logic Programs*, Tech. Report Technical Report No. 55, University of Cambridge, Computer Laboratory, 1984.

[118] S. Muggleton, *Inductive Logic Programming*, Academic Press, 1992.

[119] ______, *Inverse Entailment and Progol*, New Generation Computing **13** (1995), 245–286.

[120] S. Muggleton and L. de Raedt, *Inductive Logic Programming: Theory and Methods*, Journal of Logic Programming **19,20** (1994), 629–679.

[121] S. Muggleton and C. Feng, *Efficient Induction of Logic Programs*, Proceedings of the first Conference on Algorithmic Learning Theory, Ohmsma, 1990, pp. 368–381.

[122] S. Muggleton, King, and M.J.E. Sternberg, *Predicting protein secondary structure using inductive logic programming*, Protein Engineering **5** (1992), 647–657.

[123] D.E. Muller, *Infinite Sequences and finite Machines*, Proceedings of the 4th IEEE Symposium on Switching Circuit Theory and Logic Design, 1960, pp. 3–16.

[124] B.K. Natarajan, *On learning Boolean functions*, Proceedings of the nineteenth annual ACM conference on Theory of computing, ACM, 1987, pp. 296–304.

[125] I. Niemelä and P. Simons, *Smodels - an implementation of the stable model and well-founded semantics for normal logic programs.*, Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning, Springer Verlag, 1997, Lecture Notes in Computer Science 1265, pp. 420–429.

[126] S.-H. Nienhuys-Cheng and R. de Wolf, *Foundations of Inductive Logic Programming*, Springer Verlag, 1997.

[127] S.-H Nienhuys-Cheng, P.R.J. van der Laag, and L. van der Torre, *Constructing Refinement Operators by Decomposing Logical Impplication*, Proceedings of the 3rd Conference of the Italian Association for Artificial Intelligence, AI*IA–93, Springer Verlag, 1993, Lecture Notes in Computer Science 728, pp. 178–189.

[128] R.P. Otero, *Induction of Stable Models*, Proceedings of the 15th International Conference on Inductive Logic Programming, Springer Verlag, 2001, Lecture Notes in Computer Science 2175, pp. 193–205.

[129] M.S. Paterson and M.N. Wegman, *Linear Unification*, Journal of Computer and System Sciences **16** (1978), 158–167.

[130] P.Baumgartner and U. Furbach, *Hyper Tableaux and Disjunctive Logic Programming*, ICLP 96 Workshop on Deductive Databases and Logic Programming, vol. 295, GMD, 1996.

[131] J. Pearl, *Capacity and Error Estimates for Boolean Classifiers with limited Capacity*, IEEE Transactions on Pattern Analysis and Machine Intelligence **1** (1979), no. 4, 350–355.

[132] J. Pearl and L.G. Valiant, *Computational Limitations on Learning from Examples*, Journal of the ACM **35** (1988), no. 4, 965–984.

[133] G.D. Plotkin, *A Note on Inductive Generalization*, Machine Intelligence **5** (1970), 153–163.

[134] ______, *A Further Note on Inductive Generalization*, Machine Intelligence **6** (1971), 101–124.

[135] A.D. Procaccia and J.S. Rosenschein, *Exact VC–Dimension of Monotone Formulas*, Neural Information Processing–Letters and Reviews **10** (2006), no. 7, 165–168.

[136] R. Quinion, M.-O. Cordier, G. Garrault, and F. Wang, *Application of ILP to Cardiac Arrhythmia Characterization for Chronicle Recognition*, Proceedings of the 15th International Conference on Inductive Logic Programming, Springer Verlag, 2001, Lecture Notes in Computer Science 2175, pp. 220–227.

[137] J.R. Quinlan and R.M. Cameron-Jones, FOIL*: A midterm report*, Proceedings of the 6th European Conference on Machine Learning, Lecture Notes in Artificial Intelligence, vol. 667, Springer-Verlag, 1993, pp. 3–20.

[138] M.O. Rabin, *Decidability of second–order Theories and Automata on infinite Trees*, Transactions of the American Mathematical Society **141** (1969), 1–35.

[139] M. Reynolds and C. Dixon, *Handbook of temporal reasoning in artificial intelligence*, Foundations of Artificial Intelligence, vol. 1, ch. Theorem–Proving for Discrete Temporal Logic, Elsevier, 2005.

[140] J.A. Robinson, *A Machine-Oriented Logic Based on the Resolution Principle*, Journal of the ACM **12** (1965), no. 1, 23–41.

[141] Y. Sakai and A. Maruoka, *Learning monotone log-term DNF formulas*, Proceedings of the seventh annual conference on Computational learning theory, ACM, 1994, pp. 165–172.

[142] M. Schmidt-Schauss, *Implication of Clauses is Undecidable*, Theoretical Computer Science **59** (1988), 287–296.

[143] K. Schneider, *Verification of reactive systems – formal methods and algorithms*, Texts in Theoretical Computer Science (EATCS Series), Springer, 2003.

[144] M. Sebag and C. Rouveirol, *Constraint Inductive Logic Programming*, Advances in Inductive Logic Programming (L. De Raedt, ed.), IOS Press, 1996, pp. 277–294.

[145] E.Y. Shapiro, *An Algorithm that Infers Theories from Facts*, Proceedings of the 7th Joint Conference on Artificial Intelligence (IJCAI-81), Morgan Kaufmann, 1981, pp. 446–451.

[146] ______, *Inductive Inference of Theories from Facts*, Tech. Report Research Report 192, Yale University, 1981.

[147] J.R. Shoenfield, *Mathematical Logic*, Addison-Wesley, 1967.

[148] G.M Shutske, F.A. Pierrat, K.J. Kapples, M.L. Cornfeldt, M.R. Szewczak, F.P. Huger, G.M. Bores, V. Haroutunian, and K.L. Davis, *9–Amino–1,2.3.4–Tetrahydroacridin–1–ols: Synthesis and Evaluation as Potential Alzheimer's Disease Therapeuthics*, Journal of Medicinical Chemistry **32** (1989), no. 8, 1805–1813.

[149] A.P. Sistla, M.Y. Vardi, and P. Wolper, *The Complementation Problem for Büchi Automata with Applications to Temporal Logic*, Theoretical Computer Science **49** (1987), no. 2,3, 217–237.

[150] F. Somenzi and R. Bloem, *Efficient Büchi automata from* LTL *Formulae*, Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00), Springer Verlag, 2000, Lecture Notes in Computer Science 1855, pp. 248–263.

[151] A. Srinivasan, S. Muggleton, R.D. King, and M.J.E. Sternberg, *Mutagenesis: ILP experiments in a non-determinate biological domain*, Proceedings of the 4th International Workshop on Inductive Logic Programming (S. Wrobel, ed.), GMD-Studien, vol. 237, Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994, pp. 217–232.

[152] A. Srinivasan, S. Muggleton, R.D. King, and M.J.E. Sternberg, *The effect of background knowledge in inductive logic programming: A case study*, Tech. report, PRG-TR-9-95 Oxford University Computing Laboratory, 1995.

[153] ______, *Theories for mutagenecity: A study of first-order and feature based induciton*, Tech. report, PRG-TR-8-95 Oxford University Computing Laboratory, 1995.

[154] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, 1997.

[155] J.S. Street, *Propositional dynamic Logic of Looping and Converse is elementarily decidable*, Information and Control **54** (1982), no. 1/2, 121–141.

[156] R.E. Tarjan, *Depth first search and linear graph algorithms*, SIAM Journal of Computing **1** (1972), no. 2, 146–160.

[157] L.G. Valiant, *A Theory of the Learnable*, Communications of the ACM **27** (1984), no. 11, 1134–1142.

[158] ______, *Learning Disjunction of Conjunctions*, Proceedings of the 9th International Joint Conference on Artificial Intelligence, Morgan Kaufman, 1985, pp. 560–566.

[159] P.R.J. van der Laag, *A Most General Refinement Operator for Reduced Sentences*, Tech. Report Discussion Paper No. 123, Erasmus University Rotterdam, Faculty of Economics, 1992.

[160] P.R.J. van der Laag and S.-H Nienhuys-Cheng, *Subsumption and Refinement in Model Inference*, Proceedings of the 6th European Conference on Machine Learning, ECML–93, Springer Verlag, 1993, Lecture Notes in Computer Science 667, pp. 95–114.

[161] ______, *A Note on ideal Refinement Operators in ILP*, Proceedings of the 4th International Workshop on Inductive Logic programming, ILP–94, Gesellschaft für Mathematik und Datenverarbeitung, 1994, GMD–Studien 237, pp. 247–262.

[162] P.R.J. van der Laag and S.-H. Nienhuys-Cheng, *Existence and nonexistence of complete refinement operators*, Proceedings of the 7th International Conference on Machine Learning (F. Bergadano and L. De Raedt, eds.), Lecture Notes in Artificial Intelligence, vol. 784, Springer Verlag, 1994, pp. 307–322.

[163] P.R.J. van der Laag and S.-H Nienhuys-Cheng, *Completeness and Properness of Refinement Operators in Inductive Logic Programming*, Journal of Logic Programming **34** (1998), no. 3, 201–225.

[164] M.H. van Emden and R.A. Kowalski, *The Semantics of Predicate Logic as a Programming Language*, Journal of the ACM **23** (1976), no. 4, 733–742.

[165] J. van Heijenoort (ed.), *From Frege to Gödel: A Source Book in Mathematical Logic 1879–1931*, Harvard University Press, 1977.

[166] V.N. Vapnik and A.Y. Chervonenkis, *On the Uniform Convergence of Relative Frequencies of Events to their Probabilities*, Theory of Probability and its Applications **16** (1971), no. 2, 264–280.

[167] M.Y. Vardi and P. Wolper, *An automata-theoretic Approach to automatic Program Verification*, Proceedings of the 1st Symposium on Logic in Computer Science, Cambridge University Press, 1986, pp. 322–331.

[168] ______, *Reasoning about infinite Computations*, Information and Computation **115** (1994), no. 1, 1–37.

[169] G. Venkatesh, *A Decision Method for Temporal Logic based on Resolution*, Proceedings of the 5tc Conference on Foundations of Software Technology and Theoretical Computer Science, Springer Verlag, 1985, Lecture Notes in Computer Science 206, pp. 272–289.

[170] A.N. Whitehead and B. Russell, *Principia Mathematica*, Cambridge University Press, 1927.

[171] P. Wolper, *Temporal Logic can be more expressive*, Information and Computation **56** (1983), no. 1–2, 72–99.

[172] ______, *Constructing Automata from Temporal Logic Formulas: A Tutorial*, Lectures on Formal Methods in Performance Analysis (First EEF/Euro Summer School on Trends in Computer Science), Lecture Notes in Computer Science, vol. 2090, Springer-Verlag, July 2001, pp. 261–277.

[173] S. Wrobel, *Inductive Logic Programming for Knowledge Discovery in Databases*, Relational Data Mining (S. Dzeroski and N. Lavrac, eds.), Springer Verlag, 2001, pp. 74–104.

[174] Y. Xu, X. Song, E. Cerny, and O.A. Mohamed, *Model Checking for a First–Order Temporal Logic using Multiway Decision Graphs*, The Computer Journal **47** (2004), no. 1, 71–84.

Index

$\succcurlyeq$, 116
$\succcurlyeq_s$, 117
$\succsim$, 213

$\mathcal{A}$, 14
$\mathcal{A}(\mathrm{sig})$, 14
accessibility relation, 26
actual world, 26
adequate, 193
alphabet, 192
answer, 40, 59
  computed, 40
  correct, 40, 59
arity, 14
atom, 14
  ground, 65
  temporal, 24
automaton
  Büchi
    modified, 193
    properties, 195
  generalized Büchi, 194

background knowledge, 45
binding priorities, 10

clause, 30
  empty, 30
  ground, 66
  unit, 30
closed, 98
closedness, 98
Closure, 198
compactness theorem, 12
compatible, 133
complete, 44, 48
  $\mathcal{NP}$, 12
  co–$\mathcal{NP}$–, 12
component
  maximal strongly connected, 210
  maximal strongly connected
    non–trivial, 210
Computation Tree Logic, 264
concept, 170
concept class, 170
connectives
  first order, 27
consequence
  logical, 11, 18, 23, 28
consistent, 44
correct, 44
Ctl, 264
Ctl*, 264

decidable, 12
derivation
  SLD–, 39
derivation–step
  SLD–, 38
Det, 258
Dom, 34
domain, 34
dup, 156

EBNF, 267
equivalence
  semantical, 10
error, 172
Ev, 209
evaluation, 9
  of formulas, 16
  of terms, 16, 26
evaluation function, 9

eventualities, 200
example, 171
  classified, 171
  negative, 171
  positive, 171
Extended Backus Naur Form, 267

F, 20, 24
$\mathcal{F}$, 15
$\mathcal{F}(X)$, 8
$\mathcal{F}(\text{sig})$, 15
$\mathcal{F}$, 88
fact, 30
  PROLOG(+T)–, 55
false, 8
finiteness theorem, 11
FOLTL, 23
formula, 8, 15, 24
  atomic, 14
  closed, 67
    universally, 67
  ground, 65
FREE, 66
function
  arity–, 14

G, 20, 24
Γ_d, 220
Γ_u, 217
generalization, 46
  least, 46
  of literals, 121
  of rules, 129
goal, 30
graph
  labeled, 129
GSS, 119, 128

Herbrand–base, 66
Herbrand–universe, 66
hornclause, 30
  definite, 30

ideal, 48
ILP, 43

interpretation, 15, 25
  temporal, 21, 25

$\mathcal{L}^{\text{FoLtl}}$, 119
language
  ω, 192
  accepted, 194
lattice, 47
learner, 171
learning algorithm, 171
learning algorithm
  consistent, 172
length
  of a path, 133
letter, 192
LGS, 121, 134
list, 53
  empty, 54
  head, 54
  tail, 54
LITERAL, 136
literal, 15
  ground, 65
  negative, 30
  positive, 30
  temporal, 24
locally finite, 48
logic
  first order, 12
  predicate, 12
  propositional, 7
  temporal, 19
    first order, 19
    propositional, 19
Logic Programming, 29
  Inductive, 43
LTL$^{\text{det}}$, 256
LTL$^{n,c,t,\text{det}}$, 256
LTL^{n,c,t}, 254
LTL, 20

maximum
  of a state, 237
MD, 10

mgu, 37
minterm
  maximal, 214
MinTerms(X), 214
model, 10, 17, 22, 28
$\mathcal{L}_\mu$, 264

NDet, 258
Neg, 30
negation–normal–form, 197
NNF, 197
node, 129
  closed, 98
  completely instantiated, 100
  disjunctively expanded, 100
  father, 88
  function, 129
  predicate, 129
  son, 88
  tableaux, 88
    initial, 89
  temporal, 129

Occ, 145
occurrence, 145
operator
  Always–, 20
  Eventually–, 20
  Next–state–, 20
  Release–, 20
  Until–, 20
order
  implication, 213
  subsumption, 116
overly general, 44
overly specific, 45

P, 24
Ψ_d, 220
Ψ_u, 215
$\mathcal{P}^{\leq c,t,l,o}$, 177
$\mathcal{P}^{\leq c,t,l,o}_{\mathrm{con}}$, 183
$\mathcal{P}\mathrm{rr}^{\leq c,t,l,o}$, 183
$p(T(I))$, 70
P_M, 82

PAC–criterion, 174
PAC–learnability, 174
path, 88, 131
  closed, 98
  finite, 88
  infinite, 88
  length of, 88
  maximal, 109
  open, 98
$\Pi_\mathcal{C}$, 175
Pos (positions), 76, 156
Pos (positive literals), 30
possible worlds, 25
product construction, 201
program, 31
  Prolog(+T)–, 57
  constrained, 183
  range–restricted, 183
Prolog, 51
Prolog(+T), 52
proper, 48

quantifier
  existence, 15
  universal, 15
quantifiers, 15
quasi ordering, 46
query, 30
  Prolog(+T)–, 59

R, 20
Red, 81
reduction, 78
Refinement
  downward
    maximal, 218
  upward
    minimal, 213
refinement, 48
  n–step, 48
  1–step, 48
refinement operator, 47
  downward, 47
    for literals, 142

for rules, 150
upward, 48
for literals, 144
for rules, 157
refutation
SLD–, 39
replacement, 77
Resolution, 33
SLD–, 33
resolvent, 38
rule
$\leftrightarrow$–type, 94
$\neg$–elimination, 94
$\rightarrow$–type, 94
$\vee$–type, 94
$\wedge$–type, 93
Prolog(+T)–, 55
constrained, 183
expansion, 89
head, 56
quantifier–elimination, 94
range–restricted, 183
rewrite, 94
saturation, 94
substitution, 94
tail, 56

S^{∞}, 193
sample, 171
consistent, 171
length, 171
sample complexity, 171
satisfiability, 28
satisfiability problem, 12
satisfiable, 11, 17, 22
semantics
possible worlds, 25
Seq, 199
Set, 157
set
temporal Hintikka, 107
temporally closed, 68
shattered, 175
signature, 13
specialization, 46
greatest, 47
of literals, 119
of rules, 125
standardized apart, 58
state
temporal, 21
States, 193
states, 25
subgoal, 30
subgraph
maximal strongly connected, 210
substitution, 34
composition, 35
empty, 36
subsumption, 116, 125
symbol
flexible, 25
function, 13
predicate, 13
rigid, 25
variable, 13

$\mathcal{T}$, 14
$\mathcal{T}(\mathrm{sig})$, 14
T_P^{FoLtl}, 83
Tail, 126
TempClosure, 71
temporally closed, 68
Term, 135
term, 14
Prolog(+T), 52
ground, 65
simple, 145
Terms, 144
$\Theta_d^{\mathcal{L}}$, 143
$\Theta_d^{\mathcal{R}}$, 150
$\Theta_u^{\mathcal{L}}$, 145
$\Theta_u^{\mathcal{R}}$, 157
Θ_P, 85
$T(I)$, 69
time
branching, 19
linear, 19

too strong, 44
too weak, 44
$T(P)$, 84
TREE, 129
`true`, 8

U, 20, 24
unifiable, 37
unifier, 37
  most general, 37
universe, 16, 25
unsatisfiability, 28
unsatisfiable, 11, 17, 22

$V(\pi)$, 70
VAL, 9
valid, 11, 17, 22
validity, 28
valuation, 9
VAR, 54
variable, 8
  anonymous, 53
  free, 66
  propositional, 8
variant, 36
VCDIM, 175
VC–Dimension, 175

word
  ω, 192
  accepted, 194
  infinite, 192

X, 20, 24