KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT WETENSCHAPPEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200 A — 3001 Leuven

KNOWLEDGE REPRESENTATION AND REASONING IN INCOMPLETE LOGIC PROGRAMMING

Promotoren :
Prof. Dr. D. DE SCHREYE
Prof. Dr. J. DENEF

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de informatica

door

Marc DENECKER

September 1993

Abstract

The central research theme in logic programming is the development of a computer system in which the programmer can represent his knowledge on the problem domain in a declarative manner and which solves computational problems on the domain by means of domain independent procedures.

The thesis contains a number of contributions on the declarative and the procedural level. So far, an important problem of logic programming was that incomplete knowledge about the problem domain could not be described accurately. We show that the formalism of abductive logic programs is an expressive logic for the declarative specification of this form of knowledge. In the thesis we propose an intuitive interpretation for this formalism, we investigate the existing semantics and extend them. On the procedural level we develop NMGE, a generic procedure for model generation in first order logic with a generic equality theory, and SLDNFA, a family of abductive procedures for abductive logic programs. New theoretical relationships between abduction, deduction, model generation, consistency proving and deductive database updating are presented. The potential of the NMGE procedure for the solution of declaratively specified problems is illustrated by a simple application. The interest of abductive logic programs and SLDNFA is shown by two applications in the domain of temporal reasoning. In these applications we show how SLDNFA can be used to emulate other computational paradigms.

Preface

When I am asked how it feels to make a PhD, I tend to compare it with a long intellectual "survival journey". Indeed, it is a lonely and challenging experience. During the journey, I was on some point almost drowning in the moors, but I also enjoyed great satisfaction when exploring new untrodden abstract mathematical areas. Now that it seems that I have survived, it is time to thank many people who have contributed in one way or the other to the realisation of this thesis.

I am grateful to Prof. Yves Willems for arousing my interest in Prolog and computational logic through his courses "Theorie der programmeertalen" and "Werkseminarie logisch programmeren". Also, I thank him for the trust that he put in my colleague, Patrick Weemeeuw, and me when he engaged us for an ambitious informatics project on protein engineering. Although this first project was not really successful, the intuitions and experiences that I acquired during the project and also in the discussions with him, have formed the foundation for my research.

I would like to thank also the other members of the logic programming group, in the first place Prof. Maurice Bruynooghe, Prof. Danny De Schreye and Prof. Bart Demoen, who together with Prof. Willems, are responsible for creating a stimulating, high standard work environment. I thank them for many fruitful discussions and their constructive and, when necessary, merciless criticism on my work. I am mostly indebted to my promotor Prof. Danny De Schreye. His many critical remarks, his unrelenting perfectionism on both mathematical rigour and writing style have been vital for the realisation of this work. As Hendrik Conscience taught our people to read, Danny taught me to write. Danny made me aware of the rules underlying a good paper, the subtle battle of the author with his audience, the art of telling a coherent story and leading the reader through a labyrinth of potential objections and distracting remarks. If these qualities sometimes seem missing in this text, then I am to blame for being a bad student.

My research has benefited from multiple exchanges of ideas with Bern Martens and Lode Missiaen. Lode Missiaen raised my interest in temporal reasoning and event calculus. Eddy Bevers introduced me to the world of term rewriting. Also Robert Kowalski and Raymond Reiter have influenced my work through constructive discussions. I thank the other members of the department, in particular my

office mates, the db-group and my climbing partners for their share in the pleasant atmosphere that I experienced during the past years.

I thank Prof. Jan Denef for accepting to be co-promotor of my thesis. I am grateful to him and to the other members of my jury, Prof. Maurice Bruynooghe, Prof. Bart Demoen, Prof. Yves Willems, Prof. Marek Sergot and Prof. Els Laenens for reading my work and for providing valuable comments on it.

Finally, I want to express my special gratitude to my wife. Christel and I started a PhD almost at the same time. During six years, she was my sister-in-arms and I could count on her for support and understanding. She has defeated me by several boat's lengths by finishing her thesis six months earlier. Since then, she has taken much more than her share of the daily struggle for live on her shoulders. Now that I join her as a doctor, I confess that I could not have managed without her. Therefore, I dedicate my work to her.

August, 1993

Contents

Chapter 1

Introduction

At the origin of logic lies the goal of constructing a formal and universal language to express human knowledge [Fre67]. By its precise semantics and its universal applicability, logic was acknowledged as a tool to replace natural language in these domains where precise reasoning was of importance, as in mathematics. From the early days of computer science, the potential of logic for knowledge representation and automated reasoning was acknowledged. At the end of 50's, it was assumed that the role of logic in the computer systems of the future would be as follows. In contrast to procedural languages, logic would allow to represent any problem domain in a purely declarative way. From this declarative theory, different properties would be formally derivable through deduction. Computer systems would be based on powerful and general automated theorem provers. The input of these machines would be declarative theories describing the problem domain and formulas representing some question about a property of the problem domain. Using the declarative theory and a domain independent proof procedure, the computer would be able to answer the question.

Motivated by this perspective, an ambitious research program for automated theorem was launched in the 60's and early 70's. An important step in the evolution was the development of a revolutionary proof procedure, the *resolution* by Robinson in 1965 [Rob65]. Earlier deductive systems were based on systems of axioms and a number of primitive inference rules. Resolution replaces all inference rules and all axioms at once which makes it very well suited as the basis for mechanising deduction. Further research to more efficient control strategies over resolution eventually resulted in the development of an efficient proof procedure by Kowalski [Kow74] for definite clauses, a subclass of the first order logic. Later this procedure was called SLD-resolution [AE82], standing for *Linear Resolution with Selection function for Definite programs.* An important aspect of the same paper is that Kowalski points out that under SLD-resolution, definite clauses have a simple, elegant procedural interpretation. Kowalski's paper gave the start to

a new, more pragmatic and procedurally oriented discipline, logic programming. This discipline has extended into a wide field, where the individual topics range from theoretical foundations, over language extensions and implementation issues to a variety of applications, many of which are related to artificial intelligence and deductive databases. Common in most work is still the focus on implementation and procedural issues. This is in particular the case for Prolog, logic programming's most widely known product, which has grown to a full-fledged procedural language with a large number of primitives without declarative interpretation such as read and write commands, assert and retract and the famous *cut*. At the same time, a Prolog programming style was developed which often gives a more procedural than declarative impression.

At present, some of the original enthusiasm for the declarative representation of knowledge in logic seems to have disappeared. This may be due partly to the evolution in logic programming, in which the nature and the role of declarative knowledge representation seems somewhat blurred, and partly to the failure of the theorem proving project of the 60's and 70's. Despite important progress, current theorem provers still lack a good control of the search and suffer from a combinatorial explosion of the search space. In and outside the logic community, this failure has led some to the conclusion that a practical system based on a purely declarative logic cannot be realised, due to computational problems. An evaluation of the theorem proving project is beyond the scope of this thesis. Let me just put forward an important weakness in the project that to my opinion justifies some reserve regarding this conclusion. While during the project many efforts have been spent on the development of theorem proving techniques, little expertise has been acquired in how natural problem domains, arising in normal programming practice, can be represented in a declarative way. The negative experiences with the developed theorem provers were mostly obtained from experiments with rather formal mathematical examples of which it seems very unlikely that they are isomorphic with natural problems. An indication of this lack of experience is also seen in the fact that only during the last decade, it has become clear that problem solving in a declarative language is often not isomorphic with deduction. One observes nowadays that the role of deduction in artificial intelligence becomes smaller and smaller and that the importance of other computational paradigms is growing. This means nothing less than that one of the major premises of the theorem proving project has been superseded, namely the role of deduction as the universal computational paradigm. Because the point is important in the thesis, we illustrate it below with an example. The problem domain is the scheduling of examinations at a university. The knowledge about this domain can be represented in a number of first order logic rules. Below we give some of them.

- Each student, enrolled for the exam period, has an exam for each attended course:

$$\forall St, C : enrolled(St), attends\_course(St, C) \rightarrow \exists T : time(T) \wedge exam(St, C, T)$$

- Each student may be examined only once on a course.

$$\forall St, C, T1, T2 : exam(St, C, T1) \wedge exam(St, C, T2) \rightarrow T1 = T2$$

- Each student can attend only one exam at the same time:

$$\forall St, C1, C2, T : exam(St, C1, T) \wedge exam(St, C2, T) \rightarrow C1 = C2$$

- Each professor can attend only one type of exam at the same time:

$$\forall St1, St2, C1, C2, T : exam(St1, C1, T) \wedge exam(St2, C2, T) \wedge gives\_course(Pr, C1) \wedge gives\_course(Pr, C2) \rightarrow C1 = C2$$

- Exams can take place in the morning or the afternoon of days of the exam period:

$$\forall T : time(T) \leftrightarrow \exists D : in\_exam\_period(D) \wedge (T = morning(D) \vee T = afternoon(D))$$

- A professor can only examine when he is available:

$$\forall Pr, T, St : gives\_course(Pr, C) \wedge exam(St, C, T) \rightarrow available(Pr, T)$$

  A professor, say *y_d_willems* may have his own list of times when he is not available:

$$\forall T : \neg available(y\_d\_willems, T) \leftrightarrow (T = morning(070693) \vee T = afternoon(080693))$$

The above specification can be extended to a full specification of the problem. Evidently, it must be extended with facts describing the data, who are the students, who is enrolled for the exams, which courses the student follows, etc.. Other information may be necessary. Cardinality constraints are needed on the number of students one professor can examine at the same time; other constraints should prevent a student being examined on two successive days; other formulas require that different terms represent different individuals (the so-called unique name axioms), etc.. These extra conditions can be formulated in logic as well.

The example nicely illustrates a number of different issues in the declarative knowledge representation in logic. The problem of computing an exam scheduling is universally recognised as hard to implement in procedural languages. Therefore the example shows the potential of logic for declarative knowledge representation, it illustrates how big the gap between declarative specification and programming in procedural languages can be, but also how hard the computational problems are to solve declaratively specified problems in realistic time.

A number of advantages of declarative knowledge representation compared to procedural programming can be drawn from the example. One advantage is that the specification can easily be modified: new constraints can be added, old ones can be dropped. Using the specification, a variety of problems can be solved, each of which would require major modifications in a procedural implementation of the problem. Here the main computational problem is evidently to compute an exam schedule. An exam schedule is a set of $exam/3$ facts which satisfies all constraints. Formally, it is the subset of $exam/3$ facts occurring in a *model* of the logic specification. As a consequence, the problem of finding an exam schedule is equivalent with finding a model of this specification. In principle the problem can be solved by applying a model generation procedure on the specification.

There are other problems, of a totally different nature, which can be solved using the same specification but by applying other procedures. For example, the theory can be used to query the courses a specific student is attending, or to query whether a specific professor is available on a given day. These are deductive problems, to be solved by a theorem prover. There are other, more complex problems and procedures. For example, assume that a model/exam schedule has been computed but due to some unforeseen event, a professor cannot attend some exam. This problem can of course be solved by modifying the original specification and computing a new model. However, computing a new model may be undesirable for several reasons: this may be too costly, and it may be highly desirable that the new schedule should be as close to the old one as possible. Therefore, a better solution is to patch the existing model/schedule to accommodate for the new constraint. Here an incremental procedure is needed which, instead of recomputing a model, modifies the existing model in order to satisfy the new constraint. Such procedures are currently being developed in logic programming. This problem is congruent with the problem of intentional updates of deductive databases with integrity constraints. An intentional update procedure tries to modify a database of facts in order to satisfy a given formula called the intentional update, without violating a set of integrity constraints. If we interpret the original model as a database, the specification as a set of integrity constraints, and the new constraint as an intentional update, then an intentional update procedure will try to modify the model in order to satisfy the new constraint, without violating the original constraints. This is precisely what is needed here.

The example shows that apart from deduction, also other problem solving

paradigms such as model generation are important for solving problems using declarative specifications. During most of its history, first order logic has been associated intimately with deduction, to a degree that often it was identified with deduction. As [Ram88] observes, many logicians see logic as the study of deductive *truth preserving inference rules* (see e.g. [Rei91]). This is not the position that we take. To us a logic is in the first place a synthetic language with a precise semantics. Deduction is but one procedure to reason on logic theories. It is a technique to solve one -important- type of problems: namely the problem of determining whether some formula is implied by the theory. As illustrated by the example, there are other types of problems for which other types of procedures are required. The picture which arises here is that of a logic formalism with a declarative semantics, for which a number of problem solving procedures are available which belong to different procedural paradigms and which are related to the declarative semantics by soundness and/or completeness properties. For example, the soundness property of a deductive procedure is that it proves only valid formulas. The completeness criterion is that any valid formula can be proven by the procedure. A model generator is sound if it generates only models and it is complete if for each consistent logic theory it generates a model (or all models up to isomorphism or all minimal models up to isomorphism) (possibly in the limit).

More than model generation, one problem solving paradigm which has recently attracted a lot of attention is *abduction*. The term was introduced by the logician and philosopher C.S. Pierce (1839-1914) [Pei55]. Abduction is a procedure which can be used to generate an explanation for some observed phenomenon on the problem domain [Pop73] [Sha89]. Formally, given some theory $\mathcal{T}$ representing the problem domain and formula Q representing the observation, an abductive procedure generates a set of ground facts Δ such that the theory $\mathcal{T} \cup \Delta$ is consistent and implies Q. The exam scheduling example can be easily reformulated such that abduction applies. Indeed, let P be the theory consisting of the formulas which express the data, such as who are the students, which courses do they attend, when are professors available, etc.. and let Q be the conjunction of formulas expressing conditions for a correct exam schedule, then a set Δ of $exam/3$ facts such that $P + \Delta \models Q$ represents a correct exam schedule.

An evident but seldom observed property of abduction is that it applies only to domains with incomplete knowledge. A theory representing complete knowledge implies a formula or its negation. Abduction collapses to deduction then: any formula has either the empty set as minimal abductive solution or has no abductive solution. In the context of logic programming, abductive logic programming can be seen as a form of inductive logic programming (ILP), yet another computational paradigm. ILP generalises abduction by allowing general rules and axioms to be generated as hypotheses in Δ instead of ground facts only. Abductive reasoning can be seen a special form of ILP which applies in these situations where the domain theory $\mathcal{T}$ contains reliable general domain knowledge but lacks factual

knowledge.

During the last decade, the research program for making logic a practical computer language, which has started in the 50's with the development of more efficient theorem provers, is gradually evolving to the development of a formal declarative language with a number of different problem solving computational paradigms. This is the central theme of this thesis. The work in the thesis must be situated within the field of logic programming. At present it seems that of all the logic based research fields, logic programming provides the richest set of problem solving paradigms and most expertise in implementing these procedures. The thesis contributes to the field and to the area of artificial intelligence at different levels. At the language theoretical level, we develop a new semantics for a language extension of the logic program formalism and show relationships with FOL. At the implementation level, new algorithms are proposed for model generation and abduction; theoretical relationships are shown between computational paradigms such as model generation, abduction and deduction. At the representational level, two applications of the abductive logic program formalism and the abductive procedure are presented; both applications are concerned with temporal reasoning.

In Chapter 3, an algorithm for model generation in first order logic is presented. This algorithm extends existing model generators by an efficient treatment of equality-atoms occurring in the head of rules. At the same time, a fundamental relationship between this algorithm and abduction is shown. As was pointed out in [CTT91], abductive solutions for a definite abductive program correspond to models of the only-if part. We extend this observation by showing that the procedural semantics of abduction itself can be interpreted dually as our model generation on the only-if part.

The rest of the thesis is concerned with declarative programming in logic programming. Traditionally, logic programming is situated somewhere half-way between full declarative specification as in first order logic and procedural programming as in classical imperative languages. At this moment, there is no uniform view on Prolog formalism as a declarative logic. The declarative semantics of the formalism is currently subject of intensive study. Prolog differs from declarative logics by its fixed procedural interpretation [Kow74], originally induced on the formalism by the SLD resolution procedure. Later, negation as failure was added to the formalism, and SLD resolution was extended to SLDNF resolution. Under the *depth first, left to right* computation rule, SLDNF gives a Prolog program its interpretation as a procedural program[1]. SLDNF is a sound proof procedure with respect to all currently most accepted semantics for Prolog, and is -in general- complete wrt none of them. As a procedural language, Prolog differs from current imperative languages, by unconventional features such as unification as parameter

[1] Other procedures have been developed for the formalism of Prolog. E.g. in the context of databases, different deductive procedures were developed, in which the procedural interpretation of Prolog programs does not hold.

mechanism and backtracking.

A major restriction of Prolog from the knowledge representation point of view, is its inability to represent *incomplete knowledge* in a declarative way. Due to the closed world assumption, an atom which cannot be proved is assumed to be false. As a consequence, many problems cannot be represented declaratively in Prolog. For example, in the exam schedule specification, there may be many schedules satisfying the specification. This implies incomplete knowledge on some atoms: a given atom $exam(van\_belleghem, ips, afternoon(080693))$ may be true in one model and false in another model. This implies that neither the atom nor its negation can be proven deductively. In Prolog, a failure to prove an atom is considered as a proof for the negation of the atom. Therefore, the relation $exam/3$ cannot be implemented in Prolog as a predicate and a specification of the domain in Prolog is a much more difficult problem[2].

One domain in which the representation of incomplete knowledge in logic programming has been investigated is temporal reasoning and planning in event calculus. An event calculus is a logic program describing a temporal domain of events, states and effects of events on the state. The original event calculus only supports the prediction of a goal state, starting from a complete description of the initial state and the set of events. In planning, however, the goal is to find a set of events which realise some desired final state; the set of events is the subject of the search, and thus, a priori unknown. A solution to this problem is to extend event calculus with abduction ([Esh88], [Sha89]). In planning problems for example, the predicates which describe the events, i.e. $happens/1, act/2$ and the time precedence relation $\ll$, are abductive. An abductive solution for a goal, describing the goal state, gives a description of a set of events and their order.

The main goal of this work is to investigate the declarative semantics of abductive logic programming and the use of an abductive procedure as a computational paradigm. We present results on the declarative level, the procedural level as well as the application level.

On the declarative level, we take as a starting point, that abductive logic programs are *sets of definitions of non-abductive predicates*. We study how this is formalised in several existing semantics. The result of this study (Chapter 4) is a framework which covers and extends the three currently most widely accepted families of semantic theories for logic programming and abductive logic programming: completion semantics, stable model semantics and well-founded model semantics.

[2]This statement may surprise since it has been proven that Prolog has the expressivity of recursive functions [AN78]. Equivalently each algorithm can be implemented in Prolog. This theorem says little about the declarative capabilities of Prolog: each reasonable computer assembler language satisfies the same property. A Prolog solution will be based on a functor representation of the $exam/3$ relation and will return a list of $exam/3$ terms as an answer. Such a solution will be just the formulation of some exam schedule algorithm in Prolog and will be less declarative than the one that was presented higher. The logic of the problem domain, which was formulated naturally in our theory, will be represented in an implicit, procedural form. As a consequence, the program will lack the advantages of the declarative formulation.

The framework includes a new semantics, called justification semantics, which is proven to be an extension of the well-founded model semantics.

Two theorems are important to position abductive logic programming as a language for declarative specification. One important theorem is that under the justification semantics, a first order logic theory can be transformed to a logically equivalent abductive program. The expressivity of first order logic for declarative specification is widely recognised. This result guarantees nothing less than that the abductive program formalism is at least as expressive as first order logic. Another theorem says that an abductive procedure can be used to generate models of first order logic theories. Remarkably, this is the reverse of the result of chapter 3 on model generation. There it is shown that under some circumstances, a first order logic model generation procedure can be used to emulate abduction in logic programming.

On the procedural level, we present a new abductive procedure called SLDNFA and prove its soundness and several completeness results (chapter 5). As suggested by its name, it is a (non-trivial) extension of the above mentioned SLDNF procedure. SLDNFA differs from other existing procedures by allowing non-ground abductive atoms to be selected. This feature makes SLDNFA currently the best performing abductive procedure. Though we develop only the abductive procedure as a computational paradigm for abductive logic programs, abduction turns out to be a most flexible paradigm. We already indicated that -in principle- it can be used to emulate model generation of first order logic theories. In addition, we show how SLDNFA can be used for sound deduction in abductive logic programs, as an integrity recovery procedure and for database updating.

On the application level, two significant experiments are presented, both in the context of temporal reasoning. In one experiment (Chapter 6), a transformation is presented from a recently introduced temporal language to abductive logic programming. [GL92] introduces a new temporal language $\mathcal{A}$ which allows to represent a number of well-known benchmark problems involving incomplete temporal knowledge. They proposed a sound transformation to extended programs, programs with both negation as failure and classical negation. Both the extended program and the abductive program formalisms are language extensions of the logic programming formalism, developed for representing incomplete knowledge. The significance of our experiment is that it allows a comparison of the two formalisms.

In a second experiment (Chapter 7), we investigate event calculus for planning and temporal reasoning, continuing earlier work by [Mis91a]. SLDNFA is extended for temporal reasoning by adding a constraint solver for the time precedence relation. The resulting procedure is proven to be sound. A number of well-known benchmark problems are represented in abductive event calculus.

Chapter 2

Preliminaries

2.1 Syntax and Semantics of First Order Logic

A formal logic is a pair of its syntax and its semantics. The syntax or the formalism describes how well-formed formulas are constructed. The semantics describes the meaning of the formulas and of sets of formulas. The formalism on which we focus consists of two components: the classical first order logic formalism (FOL) and the logic programming formalism (LP). In this section we introduce the FOL syntax and its semantics. We follow mostly [Llo87].

An alphabet is the set of building blocks of a formalism. An alphabet contains six classes of symbols: variables, function symbols (including constants symbols), predicate symbols, connectives, quantifiers and punctuation symbols. Variables will be denoted by capitals X, Y, Z, functors by $f/n, g/n$ where n denotes the arity of the functor, constants are seen as 0-ary function symbols, predicate symbols are often denoted by $p/n, q/n, r/n$. The (sub-)alphabets of variables, functors, constants and predicates are user defined, the remaining classes of symbols are user-independent. Connectives are $\neg, \wedge, \vee, \leftarrow$, and $\leftrightarrow$. Quantifiers are $\exists$ and $\forall$. Punctuation symbols are ":", "(", ")", ",".

Given an alphabet, a term is defined inductively as a variable, a constant or as a syntactical object of the form $f(t_1, \ldots, t_n)$ where f/n is a functor and $t_1, \ldots, t_n$ terms. Terms are denoted by t, s. We use a shorthand notation $\overline{X}, \overline{Y}, \overline{t}, \overline{s}$ to denote tuples of variables $(X_1, \ldots, X_n)$ and $(Y_1, \ldots, Y_n)$ and tuples of terms $(t_1, \ldots, t_n)$ and $(s_1, \ldots, s_n)$. An atom of a predicate symbol p/n is of the form $p(\overline{t})$ where $\overline{t}$ is an n-tuple of terms. A FOL formula is defined inductively as an atom or as an object of the form $\neg F$, $F \wedge G$, $F \vee G$, $F \leftarrow G$, $F \leftrightarrow G$, $\exists X : F$, $\forall X : F$ where F and G denote FOL formulas and X denotes a variable. A literal is a formula of the form A or $\neg A$ where A is an atom. A variable occurs free in a formula if it occurs in a position not in the scope of a quantifier. $var(F)$ denotes the set of

free variables of F. We use the notation $F[\overline{X}]$ for a formula F to denote that the variables of $\overline{X}$ have free occurrences in F. The universal closure $\forall(F)$ of an open formula $F[\overline{X}]$ is the closed formula $\forall\overline{X} : F$. A closed or ground formula or term contains no free variables, an open formula or term contains free variables.

A class of formulas which has played an important role in logic programming is the class of the clauses. A clause is a FOL formula of form:

$$\forall\overline{X} : A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m$$

with $A_1, \ldots, A_n, B_1, \ldots, B_m$ atoms and $\overline{X}$ are the variables occurring in them. This formula is equivalent with:

$$\forall\overline{X} : A_1 \vee \ldots \vee A_n \leftarrow B_1 \wedge \ldots \wedge B_m$$

Usually, clauses are abbreviated to:

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$$

The empty clause ($n = m = 0$) is always false and is denoted by $\Box$. A negative clause has $n = 0$. A Horn clause is a clause in which $n \leq 1$. A definite clause is conventionally defined as a non-negative Horn clause. However, in section 2.3 we will distinguish between a FOL definite clause and an LP definite clause, in order to clearly distinguish the logic programming formalism from the first order logic formalism.

A FOL language $\mathcal{L}$ based on an alphabet is the set of FOL formulas constructed of the symbols of the alphabet. A FOL theory based on $\mathcal{L}$ is a set of closed FOL formulas, called its axioms. Below we will always assume that a FOL theory is recursive, i.e. there exists an algorithm which enumerates all its axioms. Mathematical logic studies larger theories but the assumption of recursiveness is a natural assumption in the context of a work about computational logic.

The goal of the *semantics* of a logic is to formalise what is the meaning of a logic theory. The search of how to formalise the meaning of the logic programming formalism is still ongoing and is one of the subjects of this thesis. In contrast, the semantics of first order logic is well-known for a long time. Central are the notions of *interpretation* and *model*; these are formally defined below. We start by introducing some convenient notations for dealing appropriately with non-Herbrand interpretations. A domain (of an interpretation) is a set. Given a domain D, we extend the conventional notions of terms and formulas by allowing domain elements to appear in them. We call them *domain terms, domain formulas.* A domain element is denoted x, y, z. A domain term or formula is called *open* if it contains free variables. Otherwise it is called *closed* or *ground.*

A closed domain formula of the form $p(t_1, \ldots, t_n)$ or $\neg p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are domain terms, is called a *fact.* In a *simple* fact, $t_1, \ldots, t_n$ are domain elements[1].

[1] A domain term (formula) can be seen, more conventionally, as a pair of a term (formula) with

An n-ary relation or a predicate can be seen as a subset of D^n or as a function from D^n to the set $\{\mathbf{f}, \mathbf{t}\}$ (where **f**,**t** stand for *false* and *true* respectively). In [Kle52], a third truth-value *undefined* or **u** was introduced. Three-valued relations are functions from D^n to $\{\mathbf{f}, \mathbf{u}, \mathbf{t}\}$. The three truth values are conventionally ordered by $\mathbf{f} < \mathbf{u} < \mathbf{t}$. Each truth value has an inverse truth value: $\mathbf{f}^{-1} = \mathbf{t}; \mathbf{t}^{-1} = \mathbf{f}; \mathbf{u}^{-1} = \mathbf{u}$.

An interpretation is a set theoretic abstraction of a problem situation. It defines a correspondence between functors and predicates with functions and relations on the problem domain.

Definition 2.1.1 (3-valued interpretation) *Given some theory $\mathcal{T}$ based on $\mathcal{L}$. A pre-interpretation I_0 of $\mathcal{L}$ consists of a domain D and a mapping of n-ary functor symbols of $\mathcal{L}$ to n-ary functions on D.*

A (3-valued) interpretation I of $\mathcal{L}$ consists of a pre-interpretation I_0 on a domain D_I, and a truth function $\mathcal{H}_I$ which maps positive simple facts to $\{\mathbf{f}, \mathbf{u}, \mathbf{t}\}$.

An incomplete interpretation for a language $\mathcal{L}$ consists of a pre-interpretation and a truth function $\mathcal{H}_I$ which is not defined for all predicate symbols of $\mathcal{L}$.

An interpretation I is 2-valued on a predicate p/n if **u** *is not in the range of the restriction of $\mathcal{H}_I$ to the facts of p/n. I is 2-valued if I is 2-valued on all predicates.*

If $\mathcal{L}$ contains "=", then we require that the interpretation of this predicate is the identity relation on D^2.

An interpretation I can be extended in a unique way to a mapping $\tilde{I}$ on all domain terms. The extension is defined by induction on the depth of the domain term:

- for any domain element x: $\tilde{I}(x) = x$
- for any f/n and domain terms $t_1, \ldots, t_n$:

$$\tilde{I}(f(t_1, \ldots, t_n) = I_0(f/n)(\tilde{I}(t_1), \ldots, \tilde{I}(t_n))$$

$\tilde{I}$ can be further extended as a mapping from positive and negative facts to simple facts: $\tilde{I}(p(t_1, \ldots, t_n)) = p(\tilde{I}(t_1), \ldots, \tilde{I}(t_n))$ and $\tilde{I}(\neg A) = \neg\tilde{I}(A)$.

A variable assignment V is defined as a set of tuples X/t with X a variable and t a domain term. The domain of V ($dom(V)$) is the set $\{X | \exists t : X/t \in V\}$. The range of V ($range(V)$) is the set $\{t | \exists X : X/t \in V\}$. This concept of variable

free variables replacing the domain elements and a variable assignment for these free variables. E.g. $p(f(X), x)$ corresponds to $(p(f(X), Y), \{Y/x\})$.

[2] Classical logic semantics allows counter-intuitive models of "=" in which "=" is interpreted by an equivalence relation. However, there is a classical result in first order logic which says that any such model can be *contracted* to an equivalent model in which "=" is interpreted by identity ([Men72], p. 82). "Equivalent" means here that any formula has the same truth value in both models. The contracted model is obtained by taking the equivalence classes of "=" as the elements of the new domain. As a consequence, allowing or disallowing these additional models has no effect on the semantics of FOL.

assignment generalises both the classical notion of variable assignment and the notion of (variable) substitution. In our terminology, a substitution is a variable assignment which assigns terms without domain elements to variables. Substitutions will as usual be denoted by θ, σ. Application of a variable assignment on terms, facts, domain formulas and sets of these are defined as usual. We denote the result as $V(F)$. $V(F)$ is called an instance of F, as usual. A variable assignment or substitution is called *grounding* for some open formula F if it assigns to each free variable a ground (domain) term.

Variable assignments (and substitutions) can be composed. The composition $V_1 o V_2$ is defined when $dom(V_1) \cap dom(V_2) = \phi$. Then $V_1 o V_2$ is the set $V_1(V_2) \cup V_1$. One easily verifies that this composition is natural: for each term t: $V_1(V_2(t)) = V_1 o V_2(t)$.

We define the semantics of FOL formulas as a (notational) variant of the 3-valued logic of [Kle52]. It is obtained by extending the truth function $\mathcal{H}_I$ of an interpretation I with domain D to all facts and closed domain formulas. For any positive fact $F = p(t_1, \ldots, t_n)$, we define $\mathcal{H}_I(F) = \mathcal{H}_I(\tilde{I}(F)) = \mathcal{H}_I(p(\tilde{I}(t_1), \ldots, \tilde{I}(t_n)))$. Using this convention, $\mathcal{H}_I$ can be extended for negative facts and closed domain formulas as follows:

$$
\begin{array}{ll}
\mathcal{H}_I(\neg F) & = \mathcal{H}_I(F)^{-1} \\
\mathcal{H}_I(F_1 \vee F_2) & = max\{\mathcal{H}_I(F_1), \mathcal{H}_I(F_2)\} \\
\mathcal{H}_I(F_1 \wedge F_2) & = min\{\mathcal{H}_I(F_1), \mathcal{H}_I(F_2)\} \\
\mathcal{H}_I(\forall X : F) & = min\{\mathcal{H}_I(\{X/x\}(F)) \mid x \in D\} \\
\mathcal{H}_I(\exists X : F) & = max\{\mathcal{H}_I(\{X/x\}(F)) \mid x \in D\} \\
\mathcal{H}_I(F_1 \leftarrow F_2) & = \mathcal{H}_I(F_1 \vee \neg F_2) \\
\mathcal{H}_I(F_1 \leftrightarrow F_2) & = \mathcal{H}_I(F_1 \wedge F_2 \vee \neg F_1 \wedge \neg F_2)
\end{array}
$$

Below, we write $M \models F$ to denote that $\mathcal{H}_M(F) = \mathbf{t}$.

Definition 2.1.2 *Let $\mathcal{T}$ be a FOL theory based on $\mathcal{L}$.*

A model M of $<\mathcal{L}, \mathcal{T}>$ is a interpretation M of $\mathcal{L}$ such that for each formula F in $\mathcal{T}$, $\mathcal{H}_M(F) \geq \mathbf{u}$. M is called a weak model if in addition there is some formula F in $\mathcal{T}$ such that $\mathcal{H}_M(F) = \mathbf{u}$. Otherwise it is a strong model.

Classical FOL is based on two-valued interpretations. For two-valued interpretations, any closed (domain) formula is either true or false. A FOL theory $\mathcal{T}$ based on $\mathcal{L}$ implies or entails some formula F iff $\mathcal{H}_M(F) = \mathbf{t}$ for each two-valued model M of $<\mathcal{L}, \mathcal{T}>$. We denote this by $<\mathcal{L}, \mathcal{T}> \models_{FOL} F$, or when $\mathcal{L}$ is clear from the context, by $\mathcal{T} \models_{FOL} F$. F is called *valid* wrt $\mathcal{T}$.

The definitions below are taken from FOL, but apply to any logic. A theory $\mathcal{T}$ based on $\mathcal{L}$ is *complete* if for each closed formula F of $\mathcal{L}$ $\mathcal{T} \models F$ or $\mathcal{T} \models \neg F$. A theory $\mathcal{T}$ is a complete axiomatisation of some problem domain, represented

by some interpretation M if $\mathcal{T}$ is complete and M is a model of $\mathcal{T}$. A sufficient condition under which $\mathcal{T}$ is a complete axiomatisation of M is when M is the only model of $\mathcal{T}$ modulo isomorphism. An isomorphism between two models is a one-to-one correspondence between the domains of the models which preserves functions and predicates (for a formal definition, see definition 3.4.8). When all models are isomorphic with M, it holds for each closed formula F that $M \models F$ iff $\mathcal{T} \models F$. Since for each F, either $M \models F$ or $M \models \neg F$, we have $\mathcal{T} \models F$ or $\mathcal{T} \models \neg F$. Hence, $\mathcal{T}$ is a complete axiomatisation of M.

Finally, we introduce a strong notion of equivalence between two theories. In the thesis we define at several occasions transformations of a theory based on some language $\mathcal{L}$ to a theory based on a language $\mathcal{L}$' which is an extension of $\mathcal{L}$. A notion of equivalence between such theories is needed. First two notions of extensions of interpretations are defined.

Definition 2.1.3 *Let $\mathcal{L}'$ be an extension of a first order language $\mathcal{L}$. Let I, I_1 be interpretations of $\mathcal{L}$, I' of $\mathcal{L}'$.*

The restriction I'' of I' to (the symbols of) $\mathcal{L}$ is the interpretation of $\mathcal{L}$ with domain D_I, with pre-interpretation I''_O which maps functors f/n of $\mathcal{L}$ to $I'_0(f/n)$ and with truth function $\mathcal{H}_{I''}$ which maps simple facts $F = p(x_1, \ldots, x_n)$ of predicates of $\mathcal{L}$ to $\mathcal{H}_{I'}(F)$.

I' is a symbol extension *of I iff the restriction of I' to $\mathcal{L}$ is identical to I.*

I_1 is a augmenting extension *of I iff I and I_1 have the same pre-interpretation and for each positive simple fact F, it holds that $\mathcal{H}_I(F) \leq \mathcal{H}_{I_1}(F)$.*

Especially in case of Herbrand interpretations, one should be careful about the notion of *restriction of an interpretation*. E.g take a language $\mathcal{L}$ with constant a and predicate $p/1$ and an extension $\mathcal{L}'$ which contains also constant b. The Herbrand interpretation $\{p(a)\}$ of $\mathcal{L}'$ is an interpretation $I = (D = \{a, b\}, \{a \rightarrow a, b \rightarrow b\}\{p(a)^{\mathbf{t}}\})$. The restriction of I to the symbols of $\mathcal{L}$ is the following non-Herbrand interpretation of $\mathcal{L}$: $(D = \{a, b\}, \{a \leftarrow a\}, \{p(a)^{\mathbf{t}}\})$ and not the following Herbrand interpretation of $\mathcal{L}$: $\{p(a)\} = (D = \{a\}, \{a \rightarrow a\}\{p(a)^{\mathbf{t}}\})$.

The following definition expresses a strong form of equivalence between the original theory and its transformation.

Definition 2.1.4 *Given is a language $\mathcal{L}$, an extension $\mathcal{L}$' of $\mathcal{L}$, a theory $\mathcal{T}$ based on $\mathcal{L}$, a theory $\mathcal{T}$' based on $\mathcal{L}$'.*

The theory $<\mathcal{L}', \mathcal{T}'>$ is an elementary extension of $<\mathcal{L}, \mathcal{T}>$ iff each model M of $<\mathcal{L}, \mathcal{T}>$ has a symbol extension M' which is a model of $<\mathcal{L}', \mathcal{T}'>$ and if, vice versa, the restriction of each model of $<\mathcal{L}', \mathcal{T}'>$ to $\mathcal{L}$ is a model of $<\mathcal{L}, \mathcal{T}>$.

Another strong form of equivalence is the notion of *conservative extension* [Sho67]: $<\mathcal{L}', \mathcal{T}'>$ is a conservative extension of $<\mathcal{L}, \mathcal{T}>$ if it holds that each formula F of $\mathcal{L}$ which is entailed by $<\mathcal{L}, \mathcal{T}>$ is entailed by $<\mathcal{L}', \mathcal{T}'>$ and vice versa,

each formula of $\mathcal{L}$ which is entailed by $<\mathcal{L}', \mathcal{T}'>$ is entailed by$<\mathcal{L}, \mathcal{T}>$. The notion of elementary extension is at least as strong as the notion of *conservative extension.*

Proposition 2.1.1 *If $<\mathcal{L}', \mathcal{T}'>$ is an elementary extension of $<\mathcal{L}, \mathcal{T}>$ then $<\mathcal{L}', \mathcal{T}'>$ is a conservative extension of $<\mathcal{L}, \mathcal{T}>$.*

Proof Assume that $<\mathcal{L}', \mathcal{T}'>$ is an elementary extension of $<\mathcal{L}, \mathcal{T}>$. We prove that it is a conservative extension of $<\mathcal{L}, \mathcal{T}>$. Assume that F is a formula based on $\mathcal{L}$ and F is entailed by $<\mathcal{L}', \mathcal{T}'>$. Take any model M of $<\mathcal{L}, \mathcal{T}>$. By definition of elementary extension, M has a symbol extension M' which is a model of $<\mathcal{L}', \mathcal{T}'>$. Clearly $\mathcal{H}_M(F) = \mathcal{H}_{M'}(F) = \mathbf{t}$. Vice versa, assume that F is entailed by $<\mathcal{L}, \mathcal{T}>$. Let M be any model of $<\mathcal{L}', \mathcal{T}'>$. By definition of elementary extension, the restriction of M to $\mathcal{L}$ is a model M' of $<\mathcal{L}, \mathcal{T}>$. Clearly, $\mathcal{H}_M(F) = \mathcal{H}_{M'}(F) = \mathbf{t}$. □

2.2 What is in a model?

A logic theory describes *knowledge* about some problem domain. The goal of a semantic theory for a logic is to formalise what is the knowledge represented by the theory. It is interesting to spend some more attention to this aspect of the FOL model theory, because it clarifies the view on interpretations, models, negation, incomplete knowledge, etc. in classical logic. Many researchers in logic programming view a model as a description of what atoms are known to be true, what atoms are known to be false and what atoms have unknown truth value. In the sequel we call this the *knowledge state semantics.* In contrast, the FOL model semantics is a *possible state* semantics: a model represents a possible state of the problem domain. Things are well-illustrated by bringing two roles on the scene: the role of the *Knowledge Engineer* and of the *System Engineer.*

The task of the Knowledge Engineer is to represent a given problem domain in a logic theory. In a first step, he must recognize the relevant objects, the relevant functions and the relevant relations between the objects of the problem domain. This boils down to making a set theoretic abstraction of the problem domain. In a second step, he chooses logical symbols to denote these relevant objects, functions and relations. This defines a logical language and, at the same time, an interpretation of the new language. This special interpretation is called the *intended interpretation.* In the third step, the Knowledge Engineer makes explicit knowledge about the problem domain by writing logical formulas (or logic programs) based on the logical language.

For the Knowledge Engineer, there are two natural quality criteria for his logic theory: correctness and completeness. The correctness criterion is that the intended interpretation should be a model, i.e. all formulas of the theory must have truth value $\mathbf{t}$ in the intended interpretation, the abstracted problem domain. In

practice, it is often impossible to check this formally, for different reasons: the mathematical construction of the intended interpretation may be too complex, or the knowledge engineer may have only incomplete knowledge on the problem domain. In principle however, the correctness of the formulas can easily be verified informally by checking the correctness of the *declarative reading* of the formulas: given the intended interpretation, any FOL formula can easily be translated to an English sentence (or dutch), called its *declarative reading*. All connectors and quantifiers in FOL have a close natural language equivalent, and there is a clear isomorphism between formal truth of a formula and the truth of its natural language translation. E.g. if $p(X, Y)$ has intended interpretation that X is a parent of Y and $q(X, Y)$ has intended interpretation that X is a grandparent of Y, then the declarative reading of the following formula:

$$\forall X, Y, Z : p(X, Y) \wedge p(Y, Z) \rightarrow q(X, Z)$$

is the (true) sentence: *for all X, Y, Z: if X is a parent of Y and Y is a parent of Z then X is a grandparent of Z*. All this is obvious, but nevertheless of crucial importance: this is a key feature which makes classical logic such a suitable medium for representing knowledge.

A second quality criterion of a theory for the Knowledge Engineer is *completeness*. A theory does not embody all knowledge of the Knowledge Engineer if he observes that his theory has a model of which he knows that it does not correspond to a possible state of the world. Therefore, the completeness criterion is that all models of the theory correspond to possible states of the world.

A totally different role is that of the System Engineer. His task is to develop domain independent procedures which use some logic theory to compute correct information. With respect to deduction, his task is to develop domain independent theorem provers which nevertheless derive true statements about the problem domain. A theorem prover derives information which is true in all models of the theory. Therefore, under the assumption that the theory is correct (that the intended interpretation is a model), any sound theorem prover derives information which is true in the problem domain, even though the System Engineer and his theorem prover have absolutely no knowledge on what is the users "external" interpretation of the symbols. This way, classical model semantics formalises in a beautiful way how at the same time a Knowledge Engineer gives some specific interpretation to the symbols of its theory, while a theorem prover does not and still can produce correct results. If the theory of Knowledge Engineer does not satisfy the completeness criterion, then a model theorem may not be able to infer information which is correct in the conception of the Knowledge Engineer. A model generator applied on such a theory may generate models which are not possible according to the Knowledge Engineer.

Some further observations are related to the content of this thesis. We already addressed the fact that the Knowledge Engineer may have only incomplete knowledge on his problem domain. Essentially, this means that his knowledge allows the

problem domain to be in more than one state. This is reflected on the formal side, by the fact that his theory will have multiple, essentially different models, set-theoretic abstractions of the possible states of the problem domain. E.g. suppose he knows that John is accountant or secretary without knowing which of the two. There will be models of his theory in which $accountant(john)$ is true and others in which $secretary(john)$ is true. In all models, $has\_job(john)$ will be true. The insight that incomplete knowledge corresponds to theories with different models, is of particular importance in our work: in chapter 4, we define a semantics for abductive logic programs in which an abductive program may have many models and prove that the formalism is as expressive as FOL for representing incomplete knowledge.

A second observation is on negation. Currently, negation is an important topic in logic programming. Recently, a logic program formalism was proposed [GL90a] which includes two types of negation; classical negation and negation by default. It is therefore interesting to look at how negation is viewed in classical logic. In classical logic, the law of the excluded middle holds:

$$\forall \overline{X} : p(\overline{X}) \vee \neg p(\overline{X})$$

This is a direct consequence of the fact that classical logic is based on two-valued interpretations. The law of excluded middle corresponds to the natural language use of negation. E.g. when John and Mary are not in the situation of being married, then John and Mary are not married. If they are not in the situation of not being married, then they are married.

The law of excluded middle gives a precise declarative characterisation of negation in FOL: a predicate symbol p/n and its negation $\neg p/n$ should only be used to represent complementary concepts: two concepts are complementary when they are not true at the same time and when one is false then the other is true. The law of excluded middle is not satisfied in three-valued interpretations. In chapter 4 we give an example where this feature of three-valued semantics is used to represent two non-complementary concepts by a predicate symbol and its negation.

We finish this section with a comment on model theory. Is model theory the only way of giving a formal account of meaning? [Rei91] expresses his doubts concerning this as follows:

> However, even given the need for semantics, the question arises why you should prefer the type of formal treatment of meaning, and truth, that is part of logic over all the possible accounts of meaning. Surely, there are alternatives, and maybe some of the alternatives are as good as the model-theoretic account of meaning given in first order logic.

There is no way to refute the existence of alternative semantic theories, but it cannot be easy to develop an alternative semantics which captures as much of the meaning of a logic theory as does model semantics. Note that at least two

elementary concepts are based directly on the notion of a model as a mathematical abstraction of (a possible state of) the problem domain: the concept of declarative reading of a formula and the related notion of intended interpretation. Of more practical importance is the role of model generation as a problem solving paradigm, as illustrated by the exam scheduling problem in chapter 1. The use of a model generator to search for possible states of the problem domain, depends crucially on the fact that a model is a mathematical abstraction of (a possible state of) the problem domain.

One might even argue to the contrary that for any formal language with a formal semantics, it should be possible to develop a model theory. Surely, any formal language has the intention to describe some external domain. A program or theory in a formal language essentially describes this domain by restricting its possible states. The notions of interpretation and model are only mathematical abstractions of the state of the problem domain and a state of the problem domain which is allowed by the program or theory. Model theory is the mathematical description of the notion of interpretation and the relationship between a program and its models.

2.3 Syntax and (some) semantics for Logic Programs

The logic programming formalism has its origin in clausal logic. Kowalski's work [Kow74] gave the start to use the definite program formalism as a procedural language. From then on, logic programming gradually moved away from classical logic, both on the syntactical level and on the semantical level. At this moment, the first order logic formalism and the logic programming formalism are really two different formalisms.

It was mentioned earlier that the original LP formalism is the definite clause formalism. It was extended a first time by allowing negative literals to appear in the body of the rules. Later it was further extended by allowing complete FOL formulas to appear in the body and with abductive predicates.

In order to distinguish explicitly LP formulas from FOL formulas, we introduce a new connective " :- " to replace the clausal operator "$\leftarrow$". A general clause is a formula of the form:

$$A \text{ :- } B_1, \ldots, B_n$$

where A is an atom and $B_1, \ldots, B_n$ are FOL formulas. A normal clause is a general clause in which $B_1, \ldots, B_n$ are literals. In a definite clause, $B_1, \ldots, B_n$ are atoms.

A general query Q is a formula of the form:

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

where $B_1, \ldots, B_n$ are FOL formulas. It denotes the formula $\forall \overline{X} : \neg B_1 \vee \ldots \vee \neg B_n$, where $\overline{X}$ are the free variables. $\neg Q$ denotes the formula $\exists \overline{X} : B_1 \wedge \ldots \wedge B_n$ (sometimes this can be confusing: in general, $\neg Q$ is the formula one is interested in and must be proved). A normal query and a definite query are defined analogously as normal and definite clauses. A definite query corresponds to a negative clause. The set of general clauses and queries defines the logic program formalism. An LP language $\mathcal{L}$ is the set of all general clauses and general queries. The logic formalism that will be used in the context of this paper is the union of FOL formalism and of the logic program formalism. When necessary, we call a language of this united formalism an LP-FOL language.

A *general definition* for a predicate p/n of $\mathcal{L}$ is a set of general clauses, each having a head of predicate p/n. Analogously, normal and definite definitions can be defined. We define a *(general) (normal) (definite) logic program* as a set of (general) (normal) (definite) definitions for predicates not including $=$. We distinguish between defined predicates having a (possibly empty) definition and undefined predicates which have no definition. Note that being an undefined predicate is not the same as having an empty definition. A logic program is *complete* if each predicate symbol of $\mathcal{L}$ except "$=$" is defined, otherwise it is *incomplete*. These definitions differ from the more conventional definitions. What conventionally is called a general program corresponds in our terminology to a complete general logic program. The conventional abductive program corresponds to an incomplete logic program and an abductive predicate corresponds to an undefined predicate. Originally, abductive logic programs were only used in the context of abductive execution. An abductive logic program was seen as a tuple of a set of normal or general clauses and a set of abductive predicates for which the program contains no definition. An abductive procedure computes for a given query Q a set of hypotheses Δ consisting of ground abductive facts such that $P + \Delta$ is consistent and $P + \Delta \models \neg Q$. However, at present, declarative semantics for abductive predicates have been defined (see chapter 4) and other types of computational paradigms can be implemented for it. Therefore, in the sequel we mostly use the term incomplete logic program.

A theory $\mathcal{T}$ based on an LP-FOL language $\mathcal{L}$ consists of a logic program $\mathcal{T}_d$ and a FOL theory $\mathcal{T}_c$, both based on $\mathcal{L}$. In the sequel, we call the FOL formulas the FOL axioms. Instead of writing a logic program $\mathcal{T}_d$ as a set of definitions (=set of general clauses), we use the conventional way of writing it as a set of clauses. The empty definition of a predicate p/n is denoted by $p(\overline{X}) \leftarrow \Box$ (instead of by ϕ). Below, a theory is given with a definition for $p/1$, the empty definition for $q/1$ and one FOL axiom:

$$\{p(a) \text{:-} p(X) \;\; ; \;\; p(X) \text{:-} q(X); \;\; q(X) \text{:-} \Box; \;\; \exists X : q(X)\}$$

In the sequel, we shall mostly focus on normal clauses. A logic program is understood to contain only normal clauses, unless explicitly mentioned.

The search for the semantics of the logic programming formalism is still ongoing. Two semantics of historical importance for LP are the least Herbrand model semantics for complete definite logic programs [vEK76] and the completion semantics for complete normal or general logic programs [Cla78]. We recall first the least Herbrand model semantics.

The Herbrand universe of $\mathcal{L}$ is the set of all ground terms of $\mathcal{L}$ and is denoted by $HU(\mathcal{L})$ or HU when $\mathcal{L}$ is obvious from the context. The Herbrand base of $\mathcal{L}$ is the set of ground atoms based on $\mathcal{L}$ and is denoted by $HB(\mathcal{L})$ or HB. A special class of interpretations are the Herbrand interpretations. A Herbrand interpretation M is an interpretation with domain $HU(\mathcal{L})$ and for each functor f/n of $\mathcal{L}$, $M(f/n)$ is the (trivial) function which maps terms $t_1, \ldots, t_n$ to the term $f(t_1, \ldots, t_n)$.

Note that for Herbrand interpretations, the notions of simple fact and ground atom coincide and that the set of atoms which are true is a subset of $HB(\mathcal{L})$. As a matter of fact, it is clear that there is a one to one correspondence between 2-valued Herbrand interpretations and subsets of $HB(\mathcal{L})$. This gives an alternative definition of a Herbrand interpretation (which is restricted to 2-valued interpretations). Using this alternative definition, the subset relation $\subseteq$ defines a partial order on the set of (2-valued) Herbrand interpretations.

The following result is due to [vEK76].

Proposition 2.3.1 *Let P be a definite program. There exists a unique (2-valued) least Herbrand model M_P of P. Or, for each (2-valued) Herbrand model M, $M_P \subseteq M$.*

Moreover, M_P is the set of valid atoms of P:

$$M_P = \{A | A \in HB(\mathcal{L}) \wedge P \models A\}$$

In the sequel, the least Herbrand model of a theory $<\mathcal{L}, P>$ is denoted by $LHM(<\mathcal{L}, P>)$. Note that it is by definition a model according to FOL semantics.

The special features of the least Herbrand model has lead van Emden and Kowalski to use it as a basis for a new semantics: the least Herbrand model semantics. Under this semantics, a definite program has one unique model, the least Herbrand model. It is important to realise that the meaning of a definite program P under least Herbrand model semantics (in the sequel, LHM semantics) differs considerably from its meaning under FOL semantics. A definite program P under FOL semantics does not imply a negative ground literal. This follows directly from the fact that the Herbrand base itself is a model of P wrt FOL semantics. On the other hand, a definite program P under LHM semantics implies lots of negative information: for each formula F (and a fortiori, each atom), P implies either F or $\neg F$. This is a trivial consequence of the fact that a definite program has only one model under LHM semantics.

This has obviously implications with respect to the declarative reading of a definite program. Consider the program P:

$p(a)$:-
$p(s(s(X)))$:- $p(X)$

where the intended interpretation is the domain of the natural numbers with a corresponding to 0, $s/1$ corresponding to the successor function and $p(X)$ representing that X is even. The declarative reading of P under FOL semantics is:

0 is an even number and
If X is an even number then $X + 2$ is an even number

This declarative reading of a formula is still correct under LHM semantics (formally this is because the least Herbrand model is a classical FOL model) but gives a highly incomplete picture of the meaning of the program. According to FOL semantics, the meaning of P is given strictly by the conjunction of these two English sentences, without excluding for example that 1 is also an even number, that 1 is equal to 0 or that there are other elements in the problem domain than the natural numbers. According to LHM semantics, P means more than that: a number which cannot be proven to be even is not even (this corresponds to the Closed World Assumption as it was called later by [Rei78b]). In addition, the domain of interpretation is isomorphic with the natural numbers, and two numbers cannot be equal.

Recall that a theory is correct from the point of view of the Knowledge Engineer if its intended interpretation is a model of the theory. Under LHM semantics, a definite program is correct iff the intended interpretation is (isomorphic with) the least Herbrand model. An implication is that a definite program cannot be used to represent incomplete knowledge in a declarative way, since it allows only one state of the problem domain. Another indication of this is that a definite program under LHM semantics is complete: every formula is either implied or its negation is implied.

The LHM semantics cannot directly be extended to normal logic programs. A restriction of the LHM semantics is that it does not work for normal and general logic programs. The first semantics developed for normal (complete) logic programs was the completion semantics of [Cla78]. In [Llo87], it was extended to general complete logic programs. In [CTT91], it was further extended to abductive logic programs. The completion of a general (abductive) logic program P is a classical logic theory, denoted *comp*(P). One part of *comp*(P) is the Free Equality theory $(FEQ(\mathcal{L}))$, also called Clark's Equality theory. It contains the standard theory of equality and some additional axioms, given by the following definitions.

Definition 2.3.1 $(EQ(\mathcal{L}))$ *The standard theory of equality for a first order language* $\mathcal{L}$*, denoted* EQ$(\mathcal{L})$*, is the following theory*[3]*:*

$\forall X : X = X$

[3]Note that all the axioms of $EQ(\mathcal{L})$ are tautologies when "=" is interpreted by identity.

$\forall X, Y : X = Y \leftarrow Y = X$
$\forall X, Y, Z : X = Z \leftarrow X = Y, Y = Z$
For each functor f/n *in* $\mathcal{L}$ $(n > 0)$*:*
$\forall \overline{X}, \overline{Y} : f(\overline{X}) = f(\overline{Y}) \leftarrow X_1 = Y_1, \ldots, X_n = Y_n$
For each predicate symbol p/n *in* $\mathcal{L}$*, different from* $=$*:*
$\forall \overline{X}, \overline{Y} : p(\overline{X}) \leftarrow p(\overline{Y}), X_1 = Y_1, \ldots, X_n = Y_n$

Definition 2.3.2 ($FEQ(\mathcal{L})$) *The Free Equality theory for a first order language* $\mathcal{L}$*, denoted* FEQ($\mathcal{L}$)*, consists of* EQ($\mathcal{L}$) *and in addition of the following axioms:*

For each functor f/n *in* $\mathcal{L}$ $(0 < n)$ *and for each* $i, 1 \leq i \leq n$*):*
$\forall \overline{X}, \overline{Y} : f(\overline{X}) = f(\overline{Y}) \rightarrow X_i = Y_i$
For each pair of different functors g/m *and* f/n *in* $\mathcal{L}$ $(n, m \geq 0)$*:*
$\forall \overline{X}, \overline{Y} : \neg g(\overline{X}) = f(\overline{Y})$
For each t*, a term which contains the variable* X*:*
$\forall (\neg t = X)$

Observe that both $EQ(\mathcal{L})$ and $FEQ(\mathcal{L})$ are Horn clause theories.

The second part of $comp(P)$ consists of the *completed definitions* of the defined predicates of P. The completed definitions are obtained in two steps. In the first step each general clause:

$$p(\overline{t}) \text{ :- } B$$

with variables $\overline{Y}$, is transformed to a formula:

$$p(\overline{X}) \text{ :- } \exists Y_1, .., Y_m : X_1 = t_1 \wedge .. \wedge X_n = t_n \wedge B$$

where $\overline{X}$ are fresh variables. Assume that in this first step for a given defined predicate p/n, we obtained the formulas $p(\overline{X}) \text{ :- } \phi_1$, ..., $p(\overline{X}) \text{ :- } \phi_k$. In the second step, these formulas are replaced by:

$$\forall (p(\overline{X}) \leftrightarrow \phi_1 \vee \ldots \vee \phi_k)$$

$comp(P)$ is a FOL theory and defines indirectly a model semantics for P. In the sequel, a model of P wrt completion semantics is meant to be a classical model of $comp(P)$.

$comp(P)$ can be seen as the conjunction of the if-part of the definitions, corresponding to P itself and the only-if part of the definitions. We define *only-if*(P) as the theory consisting of the axioms of $FEQ(\mathcal{L})$ and of the formulas of the form:

$$\forall (p(\overline{X}) \rightarrow \phi_1 \vee \ldots \vee \phi_k)$$

The completed definition of the predicate $even/1$ in the natural number example is the following:

$$\forall X : even(X) \leftrightarrow X = a \vee \exists Y : X = s(s(Y)) \wedge even(Y)$$

The declarative reading of P under completion semantics is the FOL declarative reading of the completed definition of $even/1$ and $FEQ(\mathcal{L})$ under the intended interpretation. For the natural number program P, this reading is obviously correct. It grasps some of the information content of P under LHM semantics but not all: it excludes that all odd numbers are even but does not exclude that there are other domain elements than the natural numbers for which *even* is true. Below we construct an example of such a model.

Example Define the domain of the interpretation M as the disjunct union of $\mathbb{N}$ and $\mathbb{Z}$. $M(s/1)$ is the union of the successor functions on the natural numbers and on the integer numbers. The interpretation of $even/1$ is defined as:

$$\{even(2 \times n)^{\mathbf{t}} | n \in \mathbb{N}\} \cup \{even(2 \times z + 1)^{\mathbf{t}} | z \in \mathbb{Z}\}$$

One easily verifies that this is a model of $comp(P)$.

A final comment is on the " :- " connective. We have introduced it to distinguish between the logic program clauses and the FOL formulas. In [Prz90], " :- " was introduced as a new connective in the FOL formalism with a different (FOL) meaning on 3-valued interpretations than "$\leftarrow$". In the sequel we do the same. For any interpretation I, I's truth function is defined on $F_1 \text{ :- } F_2$ in the following way:

$$\begin{aligned} \mathcal{H}_I(F_1 \text{ :- } F_2) &= \mathbf{t} \text{ iff } \mathcal{H}_I(F_1) \geq \mathcal{H}_I(F_2) \\ \mathcal{H}_I(F_1 \text{ :- } F_2) &= \mathbf{f} \text{ iff } \mathcal{H}_I(F_1) < \mathcal{H}_I(F_2) \end{aligned}$$

Note that for 2-valued interpretations the truth function for "$\leftarrow$" and for " :- " coincides.

In the same way as a definite program can be interpreted as a FOL theory of definite FOL clauses, the introduction of :- as a FOL connective implies that a logic program P can now also be interpreted as a FOL theory. It should be stressed that P interpreted as a logic program has a different semantics than P interpreted as a FOL theory.

Chapter 3

Duality of Abduction and Model Generation

3.1 Introduction

The work reported here was motivated by some recent progress made in the field of Logic Programming to formalise abductive reasoning as logic deduction [CTT91] [Bry90]. In [Kow91], Kowalski presents the intuition behind this approach. He considers the following simple definite abductive logic program:

$P = \{$ *wobbly-wheel :- flat-tyre*
wobbly-wheel :- broken-spokes
flat-tyre :- punctured-tube
flat-tyre :- leaky-valve $\}$

where the predicates broken-spokes, punctured-tube and leaky-valve are the abductive predicates. Given a query $Q = \leftarrow$ *wobbly-wheel*, abductive reasoning allows to infer the assumptions:

$S_1 = \{$ *punctured-tube* $\}$,
$S_2 = \{$ *leaky-valve* $\}$, and
$S_3 = \{$ *broken-spokes* $\}$.

These sets of assumptions are abductive solutions to the given query Q in the sense that for each S_i, we have that $P \cup S_i \models \neg Q$[1].

Kowalski points out that we can equally well obtain these solutions by deduction, if we first transform the abductive program $P \cup \{Q\}$ into a new logic theory T. The transformation consists of taking the only-if part of every definition of a

[1] In this chapter, "$\models$" should be understood as "$\models_{FOL}$".

non-abducible predicate in the Clark-completion of P and by adding the negation of Q. In the example, we obtain the (non-Horn) theory T:

$$T = \{\ \textit{wobbly-wheel} \rightarrow \textit{flat-tyre, broken-spokes}$$
$$\textit{flat-tyre} \rightarrow \textit{punctured-tube, leaky-valve}$$
$$\textit{wobbly-wheel} \leftarrow \quad \}$$

Minimal models for this new theory T are:

$$M_1 = \{\ \textit{wobbly-wheel, flat-tyre, punctured-tube}\ \},$$
$$M_2 = \{\ \textit{wobbly-wheel, flat-tyre, leaky-valve}\ \}, \quad \text{and}$$
$$M_3 = \{\ \textit{wobbly-wheel, broken-spokes}\ \}.$$

Restricting these models to the atoms of the abducible predicates only, we precisely obtain the three abductive solutions S_1, S_2 and S_3 of the original problem.

The above observation points to an interesting issue; namely the possibility of linking these dual declarative semantics by completely equivalent dual procedures. Figure 3.1 shows this duality between an SLD+Abduction tree (see [CP86]) and the execution tree of Satchmo, a theorem prover based on model generation [MB87].

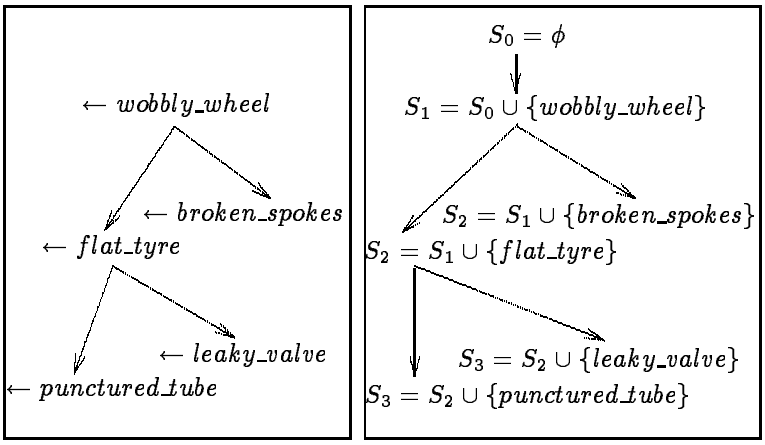

Figure 3.1: Procedural Duality of Abduction and Satchmo

Although this example illustrates the potential of using deduction or more precisely, model generation, as a formalisation of abductive reasoning, an obvious restriction of the example is that it is only propositional. Would this approach also hold for the general case of definite abductive programs? An example of a non-propositional program and its only-if part is given in figure 3.2.

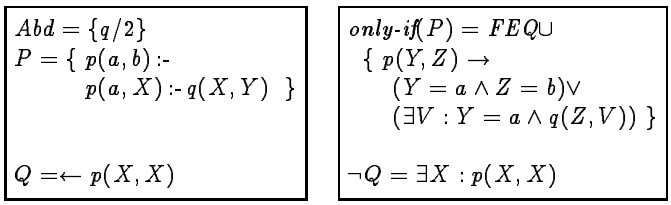

Figure 3.2: A predicate example

The theory *only-if*(P) consists not only of the only-if part of the definitions of the predicates but comprises also the axioms of Free Equality (FEQ), also known as Clark Equality [Cla78]. The abductive solutions and models of *only-if*(P) are displayed in figure 3.3.

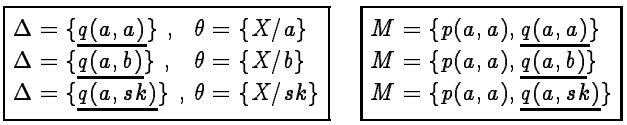

Figure 3.3: Abductive solutions and models

The duals of the abductive solutions are again identical to models of *only-if*(P). This example suggests that at least the duality on the level of declarative semantics is maintained. However, on the level of procedural semantics, some difficulties arise. An SLD+Abduction derivation tree is given in figure 3.4. After skolemisation of the residue $\leftarrow q(a,V)$, we obtain the third abductive solution.

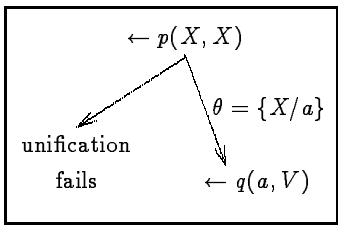

Figure 3.4: Abductive derivation tree

With respect to the model generation, a first problem is that *only-if*(P) is not clausal and Satchmo can not deal with non-clausal theories directly (without

normalisation to clausal form). Fortunately, the extension of Satchmo, Satchmo-1 [Bry90], can deal with such formulas directly. A second problem is that Satchmo and Satchmo-1 were not designed to cope with equality atoms occurring in the head: the generated models satisfy FEQ only when no equality atoms occur in the head of the rules. The solution is to treat equality as any other predicate and to add FEQ explicitly to the theory. But then a third problem arises: FEQ is not in range-restricted form. Satchmo-1 can only handle range-restricted formulas. However, any theory can be transformed to range-restricted form. After performing this transformation and without dealing with the technical details of the computation, one may obtain the computation tree as presented in figure 3.5.

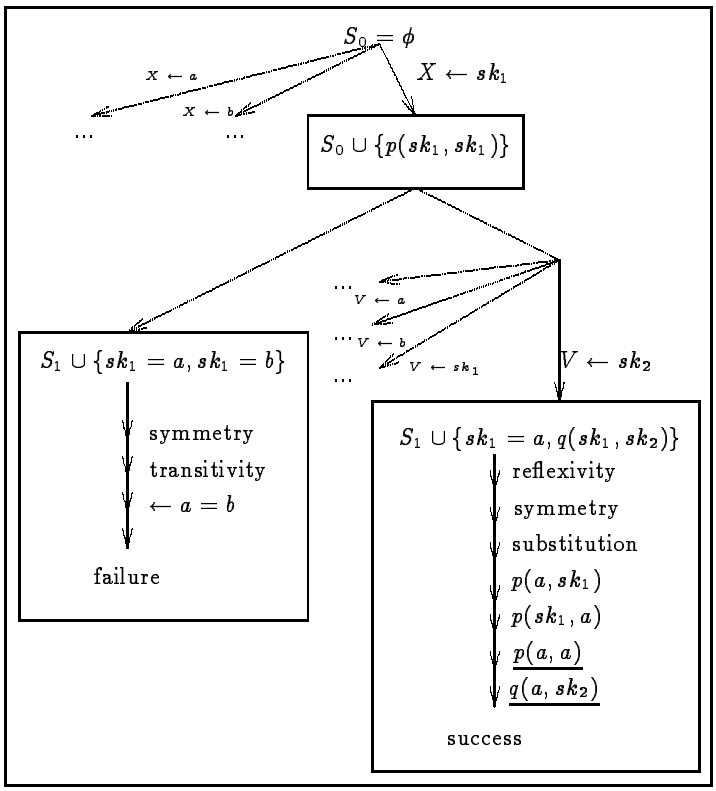

Figure 3.5: Execution tree of Satchmo-1

Globally, the structure of the SLD+Abduction tree of figure 3.4 can still be seen in the Satchmo-1-tree. Striking is the *duality* of variables in the abductive derivation and skolem constants in the model generation. However, one difference

is that the Satchmo-1 tree comprises many additional inference steps due to the application of the axioms of FEQ. In the abductive derivation the additional steps correspond to the unification operation (e.g. on both left-most branches, the failure of the unification of $\{X = a, X = b\}$ corresponds to the derivation of the inconsistency of the facts $\{sk_1 = a, sk_1 = b\}$).

Another difference is that the generated model

$$\{p(a,a), q(a,sk_2),\ p(sk_1,a), p(a,sk_1), p(sk_1,sk_1), q(sk_1,sk_2),\\ sk_1 = a, a = sk_1, a = a, sk_1 = sk_1, sk_2 = sk_2\}$$

is much larger than the model which is dual to the abductive solution. Satchmo-1 generates besides the atoms of this model also all logical implications of FEQ, comprising all substitutions of a by sk_1. It is clear that in general this will lead to an exponential explosion.

However, observe that we obtain the desired model by *contracting* sk_1 and a in the generated model. Therefore, extending Satchmo-1 with methods for dynamic contraction of equal elements would solve the efficiency problem and would restore the duality on the level of the declarative semantics.

Contraction of a model is done by taking one unique witness out of every equivalence class of equal terms and replacing all terms in the facts of the model by their witnesses. In turns out that techniques studied in Term Rewriting are useful to implement this. The idea is to consider the set of inferred equality facts as a Term Rewriting System (TRS), to transform the set to an equivalent *complete* TRS which then allows to *normalise* all terms in the model. Normalisation is a procedural way to replace terms by their witnesses.

However, a problem with the completion and normalisation procedures in Term Rewriting is that they are developed for standard equality (EQ) instead of FEQ. This has led us to develop a framework for model generation with a generic underlying equality theory (section 3.4). The framework is based on generalised notions of *completion* and *normalisation* wrt an arbitrary equality theory. Two instances of the framework have been implemented (section 3.6). One instance is a model generator for EQ, obtained by embedding existing completion and normalisation techniques from Term Rewriting in Satchmo-1. The second instance is a model generator for FEQ. It is based on the completion and normalisation procedures which we developed for FEQ (section 3.5). At first sight, these procedures may seem alien to Logic Programming, but the contrary is true: they restore the broken duality between SLD+Abduction and Satchmo-1:

- the completion procedure corresponds dually to unification.
  The dual of the mgu (by replacing variables by skolem constants) is the completion of the set of equality atoms.

- the normalisation corresponds dually to applying the mgu.

Therefore, incorporating these techniques in Satchmo-1 also restores the duality on the level of procedural semantics.

The starting goal for the research reported here, was to investigate the duality between abduction and model generation. This goal led us to a second goal, namely the extension of current techniques for model generation with efficient treatment of equality. This goal is valuable in its own right, as we argued in chapter 1: model generation is an important computational paradigm for *executing* declarative specifications.

The chapter presents a contribution to both goals. There are other spin-offs. An illustration of this is found in the context of planning as abduction in the event calculus. The event calculus contains a clause, expressing that a property holds at a certain moment if there is an earlier event which initiates this property, and the property is not terminated (clipped) in between:

$$holds\_at(P,T) \text{ :- } happens(E), initiates(E,P), \\ E <\mathrm{T}, \neg clipped(E,P,T)$$

A planner uses this clause to introduce new events which initialise some desired property. Technically this is done by first skolemising and then abducing the *happens* goal. However, skolemisation requires explicit treatment of the equality predicate as an abducible satisfying FEQ [Esh88]. The techniques proposed in this chapter allow efficient treatment of the abduced equality atoms. In chapter 5, the techniques developed in this chapter will be incorporated in an abductive procedure for efficiently dealing with abduced equality facts.

This chapter is structured as follows. In section 3.2, we present the class of theories for which the model generation is designed. Section 3.3 recalls and extends the basic concepts of Term Rewriting. In section 3.4, the framework for model generation is presented and important semantic results are formulated. In section 3.5, the duality with abductive reasoning is formalised. Section 3.6 is about the implementation of the framework. In section 3.7, we present a simple experiment. Section 3.8 discusses future and related work. A short version of this chapter appeared as [DD92a]. A long paper will be published as [DD94].

3.2 Extended programs.

In this section we introduce the formalism for which the model generation will be designed. This formalism should at least cover any theory that can be obtained as the only-if part of the definition in the Clark-completion of definite logic programs. The extended clause formalism introduced below, generalises both this kind of formulas and the clausal form.

Definition 3.2.1 *Let $\mathcal{L}$ be a first order language.*

An extended clause *or* rule *is a closed formula of the type:*

$$\forall(G_1, \ldots, G_k \rightarrow E_1, \ldots, E_l)$$

where E_i *has the general form:*

$$\exists Y_1, \ldots, Y_m : s_1 = t_1 \wedge \ldots \wedge s_g = t_g \wedge F_1 \wedge \ldots \wedge F_h$$

such that all G_i *are atoms based on* $\mathcal{L}$*, all* F_i *are non-equality atoms based on* $\mathcal{L}$*.*

Definition 3.2.2 *An extended program is a set of extended clauses.*

Interestingly, the extended clause formalism can be proved to provide the full expressivity of first order logic. Any first order logic theory can be translated to a logically equivalent extended program, in the sense that they share exactly the same models. (Recall that the equivalence between a theory and its clausal form is much weaker: the theory is consistent iff its clausal form is consistent.) We refer to appendix A for the proof of this result.

In the sequel, a theory T, based on $\mathcal{L}$, is called a *theory with equality* if it comprises $EQ(\mathcal{L})$. A theory T, based on $\mathcal{L}$ is called an *equality theory* if it is a theory with equality in which "$=$" is the only predicate symbol in all formulas except for the substitution axioms of $EQ(\mathcal{L})$.

3.3 Concepts of Term Rewriting.

The techniques we intend to develop for dealing with equality, are inspired by Term Rewriting. However, work in this area is too restricted for our purposes, because the concepts and techniques assume the standard equality theory EQ (definition 2.3.1) underlying the term rewriting. To be able to deal with FEQ, we extend the basic concepts for the case of an arbitrary underlying equality theory E. In the sequel, equality and identity will be denoted distinctly when ambiguity may occur, resp. by "$=$" and "$\equiv$". For an overview of the basic notions of TR, see [DJ89]. We recall the general ideas.

Definition 3.3.1 *Let* $\mathcal{L}$ *be a first order language.*
A reduction rule based on $\mathcal{L}$ *is of the form* $s \rightarrow t$ *where*

- s, t *are terms based on* $\mathcal{L}$*,*
- s *is not a variable,*
- *all variables in* t *are contained in* s*.*

The logical meaning of a reduction rule $s \rightarrow t$ is: "$\forall(s = t)$". Procedurally these axioms are used in a one way direction: a term containing the left-hand term of a reduction rule is *reduced* by replacing this subterm by the right-hand term.

Definition 3.3.2 *A Term Rewriting System (TRS) based on $\mathcal{L}$ is a finite set of reduction rules based on $\mathcal{L}$.*

In what follows, Term Rewriting Systems are denoted by the symbols γ and δ. With a Term Rewriting System, a reduction relation between terms can be associated.

Definition 3.3.3 *Given a Term Rewriting System γ its reduction relation $\stackrel{\gamma,\mathcal{L}}{\longrightarrow}$ is a binary relation on the set of all terms based on $\mathcal{L}$ such that $t_1 \stackrel{\gamma,\mathcal{L}}{\longrightarrow} t_2$ iff there exists a reduction rule $s \rightarrow t$ and a substitution θ such that $\theta(s)$ occurs in t_1 and t_2 is obtained by replacing $\theta(s)$ by $\theta(t)$.*

A Term Rewriting System can be seen both in a procedural way as a set of reduction rules $s \rightarrow t$ and in a declarative way as a theory of equality axioms "$\forall(s = t)$". In the sequel, γ will be used in both ways; the context will determine how to interpret it.

Recall that Herbrand Universe and Herbrand Base based on a language $\mathcal{L}$ are denoted by $HU(\mathcal{L})$, $HB(\mathcal{L})$ resp. and that the least Herbrand model of a theory T based on $\mathcal{L}$ is denoted $LHM(<\mathcal{L}, T>)$.

Definition 3.3.4 *The reflexive, symmetric, transitive closure of $\stackrel{\gamma,\mathcal{L}}{\longrightarrow}$ is denoted $\stackrel{\gamma,\mathcal{L}}{\longleftrightarrow}^*$. It's restriction to the set of ground terms $HU(\mathcal{L})$ is denoted $\stackrel{\gamma,\mathcal{L}}{=\!=}$.*

Observe that $EQ(\mathcal{L}) + \gamma$ is a definite program and hence has a least Herbrand model [vEK76]. This is the set of all ground atoms based on $\mathcal{L}$ which are logical consequences of $EQ(\mathcal{L}) + \gamma$. The following property indicates the relationship with $\stackrel{\gamma,\mathcal{L}}{=\!=}$.

Proposition 3.3.1 $\stackrel{\gamma,\mathcal{L}}{=\!=}$ *is* $LHM(<\mathcal{L}, \mathrm{EQ}(\mathcal{L}) + \gamma>)$

Proof It is easy to see that $\stackrel{\gamma,\mathcal{L}}{=\!=}$ satisfies all the axioms of $EQ(\mathcal{L}) + \gamma$. Since it is a Herbrand interpretation, $LHM(<\mathcal{L}, EQ(\mathcal{L}) + \gamma>) \subseteq \stackrel{\gamma,\mathcal{L}}{=\!=}$. Vice versa, the interpretation of "$=$" in $LHM(<\mathcal{L}, EQ(\mathcal{L}) + \gamma>)$ is reflexive, symmetric and transitive and if $s \stackrel{\gamma,\mathcal{L}}{\longrightarrow} t$ then $s = t \in LHM(<\mathcal{L}, EQ(\mathcal{L}) + \gamma>)$. Hence, $\stackrel{\gamma,\mathcal{L}}{=\!=} \subseteq LHM(<\mathcal{L}, EQ(\mathcal{L}) + \gamma>)$. □

A Term Rewriting System associates with every term a reduction tree: a tree of terms such that t_2 is a son of t_1 iff $t_1 \stackrel{\gamma,\mathcal{L}}{\longrightarrow} t_2$. Such a tree corresponds to all possible reductions of the root term. A *Noetherian* Term Rewriting System has

the property that each reduction tree is finite. A *confluent* or *Church-Rosser* Term Rewriting System has the the property that for any pair of nodes in any reduction tree, there exist paths leaving from this nodes, and leading to an identical term. From the procedural point of view, a *Noetherian* and *Church-Rosser* TRS satisfies the desirable property that the reduction tree of any term t is finite and that all its leaves are labelled by the same term, called the normal form of t and denoted $\gamma(t)$. This *normalisation* operation can be extended to atoms, formulas and sets of these in a natural way. A term which cannot be reduced any further is called *normal*. In Term Rewriting, a noetherian and Church-Rosser TRS is called *complete*. Below we extend this concept for an underlying equality theory E.

Definition 3.3.5 *Let E be an equality theory based on a language $\mathcal{L}$, γ a Term Rewriting System based on $\mathcal{L}$.*

γ is complete wrt to E iff γ is noetherian and Church-Rosser and, moreover, for each language $\mathcal{L}'$ extending $\mathcal{L}$ with constants, $<\mathcal{L}', E+\gamma>$ has a least Herbrand model, which is the set of ground atoms $s=t$ constructed from terms in $HU(\mathcal{L}')$ such that $\gamma(s)\equiv\gamma(t)$.

The introduction of the extension $\mathcal{L}'$ of $\mathcal{L}$ in the third condition assures that the property of being complete is language independent. This will prove to be important for the remainder of the chapter, because during model generation skolem constants are introduced dynamically.

This definition extends the normal definition in Term Rewriting by the third condition. However, for $E = EQ$, it was proved in [Hue80] that this condition is implied by the Noetherian and Church-Rosser properties (see also Proposition 3.3.2(a)). This is not the case for an arbitrary equality theory (as FEQ). For example take any language comprising constants a and b and take $E = FEQ(\mathcal{L})$. Define $\gamma = \{a \rightarrow b\}$. γ is noetherian and Church-Rosser but $FEQ+\gamma$ is inconsistent and hence has no least Herbrand model. In section 3.5, we prove that $FEQ(\mathcal{L})$ has only trivial complete Term Rewriting Systems, in the sense that for each reduction rule $s \rightarrow t$ of a complete Term Rewriting System s is a constant of $\mathcal{L}' \setminus \mathcal{L}$.

Much work in Term Rewriting concentrates on complete TRSs (with of course EQ as underlying equality theory). One of the central themes in TR, is the *validity problem*: given some TRS γ and terms s, t, decide whether $EQ(\mathcal{L}) + \gamma \models \forall(s = t)$. In general, the validity problem is undecidable but for a complete TRS γ, it is decidable since $EQ(\mathcal{L}) + \gamma \models \forall(s = t)$ if and only if $\gamma(s)\equiv\gamma(t)$. This interesting result has motivated the research in TR to develop methods to transform a TRS into a logically equivalent (wrt EQ) but complete TRS. This operation is called the *completion*. The best-known and oldest completion algorithm was proposed by Knuth and Bendix [KB70]. The problem of finding a complete TRS for a given set of equations wrt EQ, is in general unsolvable (otherwise, the validity problem could be solved). However, the completion of a *ground TRS* (wrt EQ) can be computed [Der87]. That suffices for our purposes, since during model generation only ground

instances of rules are applied, hence the equality sets to be completed are always ground.

Remember from section 3.1, that our main goal was to introduce *dynamic contraction* during model generation. The main reason why here we introduce the notion of a complete TRS is not to solve some validity problem but because a complete TRS γ allows to contract the partially constructed model. Indeed, because two terms equal wrt $EQ(\mathcal{L}) + \gamma$ have the same normalisation, and since a term and its normalisation occur obviously in the same equivalence class, it follows directly that each equivalence class contains precisely one normal term. Hence the normal terms can be taken as the unique witnesses, and normalisation is the procedure to contract models by replacing terms by their unique witnesses.

In the following proposition, some basic properties of complete TRSs are explored.

Proposition 3.3.2 *(a) if γ is noetherian and Church-Rosser, then γ is complete wrt* EQ($\mathcal{L}$).

(b) if γ is complete wrt E then γ is complete wrt EQ($\mathcal{L}$).

(c) If γ is complete wrt E then for each $\mathcal{L}'$ extending $\mathcal{L}$ with constants:

$$LHM(<\mathcal{L}', E + \gamma>) = LHM(<\mathcal{L}', \mathrm{EQ}(\mathcal{L}) + \gamma>)$$

Proof For a proof of (a), define $S = \{s = t | s, t \in HU(\mathcal{L}')$ and $\gamma(s) \equiv \gamma(t)\}$. We show that S is $\overset{\gamma,\mathcal{L}'}{=\!=}$. First observe that if $s \overset{\gamma,\mathcal{L}'}{\longrightarrow} t$ then the reduction tree of t occurs in the reduction tree of s, so $\gamma(s) \equiv \gamma(t)$. Hence S subsumes $\overset{\gamma,\mathcal{L}'}{\longrightarrow}$. Further on, it is trivial to see that S defines a reflexive, symmetric and transitive relation, hence $\overset{\gamma,\mathcal{L}'}{=\!=} \subseteq S$.

Vice versa, clearly each atom $s = \gamma(s)$ is an element of $\overset{\gamma,\mathcal{L}'}{=\!=}$. Because of this and of the symmetry and transitivity of "=" in $\overset{\gamma,\mathcal{L}'}{=\!=}$, for any pair of terms s, t based on $\mathcal{L}'$, $\gamma(s) \equiv \gamma(t)$ implies $s = t \in \overset{\gamma,\mathcal{L}'}{=\!=}$. So, $S \subseteq \overset{\gamma,\mathcal{L}'}{=\!=}$.

Since by Proposition 3.3.1, $\overset{\gamma,\mathcal{L}'}{=\!=}$ is $LHM(<\mathcal{L}', EQ(\mathcal{L}) + \gamma>)$, we find that $s = t \in LHM(<\mathcal{L}', EQ(\mathcal{L}) + \gamma>)$ iff $\gamma(s) \equiv \gamma(t)$. Hence, γ is complete wrt $EQ(\mathcal{L})$.

Item (b) is a direct consequence of (a): a complete TRS γ wrt E is noetherian and Church-Rosser, so by (a) γ is complete wrt $EQ(\mathcal{L})$.

Item (c) follows also directly from (a). If γ is complete wrt E then by definition $LHM(<\mathcal{L}', E + \gamma>) = S$. By (a), we have that

$$S = LHM(<\mathcal{L}', EQ(\mathcal{L}) + \gamma>)$$

□

Definition 3.3.6 *A completion of a TRS γ wrt $<\mathcal{L}, E>$ is:*

- $\{\Box\}$ *if* $<\mathcal{L}, E+\gamma>$ *is inconsistent*
- *a complete TRS* γ_c *based on* $\mathcal{L}$*, such that* $<\mathcal{L}, E> \models \gamma \leftrightarrow \gamma_c$

We denote the completion of γ by TRS-comp(γ).

Our framework for model generation is developed for logical theories consisting of two components, an extended program P and an *underlying* equality theory E. This distinction reflects the fact that the model generation mechanism applies only to the extended clauses of P, while E is dealt with in a procedural way, using completion and normalisation. E has to satisfy the conditions of the following definition.

Definition 3.3.7 *An* equality theory with completion E *based on a language* $\mathcal{L}$*, is a clausal equality theory equipped with a completion procedure which for each ground TRS based on an extension of* $\mathcal{L}$ *by constants, produces a ground completion.*

Some remarks are of interest. The condition for an equality theory of having a completion procedure is very restrictive. An equality theory E which does not consists purely of Horn clauses has in general not a least Herbrand model, hence it cannot have a completion procedure. Even Horn equality theories will in general not have a completion procedure. For example, let γ be a non-ground Term Rewriting System $\{f(X) \rightarrow g(X)\}$ based on the language with functors $f/1, g/1, b/0$. This is a complete Term Rewriting System wrt EQ. Suppose that $EQ+\gamma$ has completion procedure TRS-comp. By definition TRS-comp(ε) is a ground term rewrite system δ such that for each language extension $\mathcal{L}'$ of $\mathcal{L}$, $LHM(<\mathcal{L}', EQ(\mathcal{L})+\gamma>) = LHM(<\mathcal{L}', EQ(\mathcal{L})+\delta>)$. In general, this is impossible due to the fact that δ is ground and γ is not. Indeed, let $\mathcal{L}$' be a language extension of $\mathcal{L}$ containing a new constant a. $LHM(<\mathcal{L}', EQ(\mathcal{L})+\gamma>)$ contains $f(a) = g(a)$. On the other hand, it is clearly impossible to rewrite $f(a) = g(a)$ to identical terms by a ground Term Rewriting System which is based on $\mathcal{L}$.

As far as we know, no general technique exists to check whether an equality theory has a completion procedure. For an equality theory with completion, the completion procedure depends totally on the equality theory. In this chapter we present two theories with completion with totally different completion procedures. In section 3.5, we will prove that $FEQ(\mathcal{L})$ is an equality theory with completion and that its completion procedure is dual to the unification procedure. As indicated earlier, $EQ(\mathcal{L})$ has also a completion procedure which is a variant of the Knuth-Bendix procedure. Hence, $EQ(\mathcal{L})$ is an equality theory with completion.

Another concept taken from Term Rewriting is *E-unification.*

Definition 3.3.8 *Let t, s be terms, γ a TRS. An E-unifier θ of t and s wrt γ is a substitution such that* $\mathrm{EQ}+\gamma \models \forall(\theta(s) = \theta(t))$. *An E-unifier of atoms $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$ is an E-unifier of the tuples $(t_1, s_1), \ldots, (t_n, s_n)$. An E-unifier of a set of pairs of atoms is an E-unifier of all the pairs of atoms.*

3.4 A framework for Model Generation

Informally a model generator constructs a sequence[2] $(Cl_d, j_d)_1^n$, where Cl_d is the ground instance of a rule applied at the d-th inference step, and j_d the index indicating the conclusion of Cl_d that was selected, an increasing sequence of sets of *asserted* ground facts $(M_d)_0^n$ of non-equality predicates, a sequence of complete Term Rewriting Systems $(\gamma_d)_0^n$, each of which is equivalent with the set of *asserted* equality facts, and an increasing sequence of sets of skolem constants $(Sk_d)_0^n$, obtained by skolemising the existentially quantified variables.

Below, a substitution is called normal wrt some TRS γ if it assigns normal terms to each variable. The normalisation of a substitution θ is the substitution obtained by normalising all right-hand terms in θ. We denote the normalised substitution by $\gamma(\theta)$. An instance of an extended clause is obtained by applying a grounding substitution for the extended clause. A normal instance is an instance obtained by applying a normal substitution. Note that this does not imply that all terms in the normal instance are normal. Only the terms assigned to variables are normal.

Definition 3.4.1 *Let $\mathcal{L}$ be a language, $\mathcal{L}_{sk}$ an infinite countable alphabet of skolem constants, not occurring in $\mathcal{L}$, T a theory based on $\mathcal{L}$ and consisting of an extended program P and of an equality theory E with completion, equipped with completion procedure TRS-comp.*

A Nondeterministic Model Generator with Equality *(NMGE) K is a tuple of four sequences $(Sk_d)_0^n$, $(M_d)_0^n$, $(\gamma_d)_0^n$ and $(Cl_d, j_d)_1^n$ where $n \in \mathbb{N} \cup \{\infty\}$. The sequences satisfy the following conditions:*

1. $M_0 = Sk_0 = \{\}; \gamma_0 =$ *TRS-comp*$(\{\})$
2. *For each d such that $0 < d \leq n$, Cl_d, j_d, Sk_d, M_d and γ_d are obtained from Sk_{d-1}, M_{d-1} and γ_{d-1} by applying the following steps:*
   (a) *Selection of rule and conclusion*
   Select nondeterministically a rule $C = G_1, \ldots, G_k \rightarrow E_1, \ldots, E_l$ of P and a substitution θ such that the following conditions hold:
   - *θ is a grounding substitution of C, normal wrt γ_{d-1} and based on $\mathcal{L} + Sk_{d-1}$.*
   - *there exist atoms $A_1, \ldots, A_k$ from M_{d-1} such that θ is an E-unifier of the set $\{(G_1, A_1), \ldots, (G_k, A_k)\}$ wrt γ_{d-1}.*

[2] $(A_d)_i^n$ denotes a sequence $(A_i, \ldots, A_n)$.

Define Cl_d as $\theta(C)$. If $l = 0$, define $Sk_d = \{\}$, $M_d = \gamma_d = \{\Box\}$ and $n = d$.
Otherwise, select nondeterministically a conclusion $\theta(E_j)$ from the head of Cl_d. Define $j_d = j$. We say that the rule Cl_d applies with its j_d'th conclusion.

(b) Skolemisation
Let $\theta(E_{j_d})$ be of the form:

$$\exists Y_1, \ldots, Y_m : s_1 = t_1 \wedge \ldots \wedge s_g = t_g \wedge F_1 \wedge \ldots \wedge F_h$$

Replace $Y_1, \ldots, Y_m$ by fresh skolem constants $sk_1, \ldots, sk_m$ from the language $\mathcal{L}_{sk} \backslash Sk_{d-1}$. Define $Sk_d = Sk_{d-1} \cup \{sk_1, \ldots, sk_m\}$

(c) Completion
Define $\gamma_d = TRS\text{-}comp(\gamma_{d-1} + \{s_1 = t_1, \ldots, s_g = t_g\})$. If γ_d is $\{\Box\}$ then define $M_d = \{\Box\}$ and $n = d$.

(d) Normalisation+Assertion
Define $M_d = \gamma_d(M_{d-1} \cup \{F_1, \ldots, F_h\})$, obtained by computing the normal form of all facts in these sets.

K is failed if n is finite and $\gamma_n = M_n = \{\Box\}$. This situation occurs when Cl_n is a negative clause, or when $E + \gamma_{n-1} + \{s_1 = t_1, \ldots, s_g = t_g\}$ is inconsistent.
If K is not failed then K is called successful.

Notice that (a) requires an E-unifier θ of the body of the rule C and facts of M_{d-1}. In Proposition 3.5.3, we will show that with FEQ as underlying equality theory with completion, E-unification collapses to unification, i.e. θ is an E-unifier iff θ is a unifier.

Not all NMGEs generate models of $P + E$. For example, the empty NMGE (({}), ({}), (TRS-comp({})), ()) trivially satisfies the definition of an NMGE, but will not generate a model if P contains one positive extended clause, i.e. an extended clause with empty body. In that case the empty NMGE is an example of an unfair NMGE: there exists a rule with a true body, but which is never applied. Only *fair* NMGEs generate models:

Definition 3.4.2 *A NMGE K is fair iff K is failed or else if the following condition is satisfied: If $Cl = G_1, \ldots, G_k \rightarrow E_1, \ldots, E_l$ is a ground instance of a rule of P, and there exists a d such that Cl is based on $\mathcal{L} + Sk_d$ and the body of Cl holds in LHM_d then there exists a d' such that $E_1 \vee \ldots \vee E_l$ holds in $LHM_{d'}$.*

Definition 3.4.1 does not exclude that some NMGE continues to apply the same rule an infinite number of times. From a procedural point of view, it is uninteresting to apply a rule whose head is satisfied in LHM_{d-1}. NMGEs which apply only rules when necessary are called *nonredundant*.

Definition 3.4.3 *An NMGE is redundant iff at least one rule is applied (say at step d) which is satisfied in* LHM_{d-1}.

Example Take EQ as underlying equality theory with completion (definition 2.3.1) and consider the following theory P:

$$\rightarrow a = f(a)$$
$$p(X) \rightarrow a = X$$
$$\rightarrow p(b)$$

An NMGE is obtained as follows. In the first step, the first rule is selected. We have $\theta = \varepsilon, Cl_1 = (\rightarrow a = f(a))$ and $j_1 = 1$. The completion of $a = f(a)$ is computed by applying Knuth-Bendix completion [KB70]. This returns $\gamma_1 = \{f(a) \rightarrow a\}$. The sets M_1 and Sk_1 remain empty.

In the second step, the third rule is selected. We have again $\theta = \varepsilon, Cl_2 = (\rightarrow p(b))$ and $j_2 = 1$. γ_2 is identical to γ_1. $p(b)$ is in normal form, and $M_2 = \{p(b)\}$. Sk_2 remains empty.

In the third step, the second rule is applied. We have $\theta = \{X/b\}, Cl_3 = (p(X) \rightarrow a = X)$ and $j_3 = 1$. Now, we must compute the completion of $\{f(a) = a, a = b\}$. A solution is $\gamma_3 = \{f(a) \rightarrow a, b \rightarrow a\}$. With this TRS, M_2 is normalised to $M_3 = \{p(a)\}$. Sk_3 is still empty.

At this point, all rules are satisfied. We obtain a fair NMGE which generates the finite model $(D = \{a\}, \{a \rightarrow a, b \rightarrow a, f(a) \rightarrow a\}, \{a = a, p(a)\})$. Notice that a model generator without special treatment for equality will loop on $EQ + P$. During this loop, an infinite number of facts will be derived: for each n and m: $p(f^n(a))$, $p(f^n(b))$, $f^n(a) = f^m(b)$, etc.. are logical implications and will be derived.

Now assume that we add the axiom $p(f(f(f(f(f(a))))))) \rightarrow$. The previous NMGE must be extended by a fourth step. In this fourth step, the E-unifier between $p(f^5(a))$ and $p(a)$ wrt γ_3 is computed. The empty substitution is an E-unifier between these atoms, and we obtain failure. Notice that a model generator without special treatment of equality will also eventually stop, but this will last until $p(f^5(a))$ is derived by application of the axioms of EQ. In general, a high number of other useless atoms will be derived before.

Finally, observe that if FEQ was the underlying equality theory, then failure would occur when the rule $\rightarrow a = f(a)$ is selected. This atom is inconsistent with the occur-check axioms.

Example Take FEQ as underlying equality theory and consider the following theory P.

$\exists X : p(f(h(X), X)) \wedge q(X)$
$p(X) \rightarrow (\exists Z : X = f(Z, g(a)))$
$q(g(Z)) \rightarrow$

An NMGE selects first the first extended clause as Cl_1. We have $\theta = \varepsilon$ and $j_1 = 1$. The variable X is skolemised and the two atoms are asserted. This yields $M_1 = \{p(f(h(sk_1), sk_1)), q(sk_1)\}, \gamma_1 = \varepsilon, Sk_1 = \{sk_1\}$.

In the second step, the second rule is selected as Cl_2. We have:

$$\theta = \{X/f(h(sk_1), sk_1)\} \text{ and } j_2 = 1$$

Z is skolemised to sk_2 and we derive the equality atom $f(h(sk_1), sk_1) = f(sk_2, g(a))$. The completion of this atom is obtained by applying a dual form of unification; this yields:

$$\gamma_2 = \{sk_1 \rightarrow g(a), sk_2 \rightarrow h(g(a))\}$$

After normalisation, we obtain:

$$M_2 = \{p(f(h(g(a)), g(a))), q(g(a))\}, Sk_2 = \{sk_1, sk_2\}$$

In the third step, the third rule is selected as Cl_3. We have $\theta = \{Z/g(a)\}\}$ and $j_3 = 0$. Failure occurs.

One remark to be made here is that since the language comprises the functor $f/1$, FEQ comprises an infinite number of disequality axioms. Hence, it is impossible to use a model generator without special treatment of FEQ. A second remark is that the above theory is consistent under EQ. Indeed, a completion under EQ of the equality fact $f(h(sk_1), sk_1) = f(sk_2, g(a))$ under EQ is $\{f(h(sk_1), sk_1) \rightarrow f(sk_2, g(a))\}$. From this TRS, $sk_1 = g(a)$ cannot be derived. Therefore, the third rule cannot be applied.

Below, LHM_d denotes the least Herbrand model of $< \mathcal{L} + Sk_d, EQ(\mathcal{L}) + M_d + \gamma_d >$.

Proposition 3.4.1 *$(LHM_d)_0^n$ is a monotonically increasing sequence.*

The proof of this proposition uses the next lemma.

Lemma 3.4.1 *Let E be an equality theory with completion, based on $\mathcal{L}$. Let γ be a Term Rewriting System, $\delta = TRS\text{-}comp(\gamma)$, M a set of ground non-equality atoms, normal wrt δ, and γ, δ and M based on $\mathcal{L} + Sk$.*

For each extension $\mathcal{L}'$ of $\mathcal{L} + Sk$ by constants, the following equalities hold:

(a) $LHM(<\mathcal{L}', E + \gamma + M>) = LHM(<\mathcal{L}', E + \delta + M>)$

(b) $LHM(<\mathcal{L}', E + \delta + M>) = LHM(<\mathcal{L}', \mathrm{EQ}(\mathcal{L}) + \delta + M>)$

(c) $LHM(<\mathcal{L}', \mathrm{EQ}(\mathcal{L}) + \delta + M>) \stackrel{\delta, \mathcal{L}'}{=\!=} \cup \ \{A | A \in HB(\mathcal{L}') \text{ and } \delta(A) \in M\}$

Proof The lemma follows straightforwardly from the equivalence of γ and δ wrt E, Proposition 3.3.2 and Proposition 3.3.1 and the substitution axioms for the predicates. □

Proof (of Proposition 3.4.1)

We have that:

$$\begin{aligned}
LHM_d &= LHM(<\mathcal{L} + Sk_d, EQ(\mathcal{L}) + M_d + \gamma_d>) \\
&\subseteq LHM(<\mathcal{L} + Sk_{d+1}, EQ(\mathcal{L}) + M_d + \gamma_d>) \\
&= LHM(<\mathcal{L} + Sk_{d+1}, E + M_d + \gamma_d>) \quad \text{(Lemma 3.4.1(b))} \\
&\subseteq LHM(\mathcal{L} + Sk_{d+1}, E + M_d + \{F_1, \ldots, F_h\} + \gamma_d + \\
&\qquad \{s_1 = t_1, \ldots, s_g = t_g\} \\
&= LHM(<\mathcal{L} + Sk_{d+1}, E + M_{d+1} + \gamma_{d+1}>) \\
&\qquad \text{(Lemma 3.4.1(a))} \\
&= LHM_{d+1} \quad \text{(Lemma 3.4.1(b))}
\end{aligned}$$

□

An NMGE performs a fixpoint computation, the result of which can be seen as an interpretation of the language $\mathcal{L}$ and, as we later show, as a model of $<\mathcal{L}, P + E>$.

Definition 3.4.4 *The skolem set used by an NMGE K is $\cup_0^n Sk_d$ and is denoted by $Sk(K)$. $\cup_0^n LHM_d$ defines a Herbrand interpretation for the language $\mathcal{L} + Sk(K)$, and is denoted $K\uparrow$. The non-Herbrand interpretation of $\mathcal{L}$ obtained by restricting $K\uparrow$ to the symbols of $\mathcal{L}$ is denoted by $K\uparrow_{\mathcal{L}}$ and is defined as follows:*

- *domain: $HU(\mathcal{L} + Sk(K))$*
- *for each functor f/n of $\mathcal{L}$ ($n \geq 0$): $K\uparrow_{\mathcal{L}}(f/n)$ is the function which maps terms $t_1, \ldots, t_n$ of $HU(\mathcal{L} + Sk(K))$ to the term $f(t_1, \ldots, t_n)$.*
- *for each predicate of $\mathcal{L}$: $K\uparrow_{\mathcal{L}}(p/n)$ is the set of $p(t_1, \ldots, t_n)$ facts in $K\uparrow$.*

Corollary 3.4.1 *If K is a finite successful NMGE of length n, then $K\uparrow = LHM_n$*

The following proposition establishes a number of basic results which will be used frequently in this and the following section. The first result assures us that if the head of some instance of a rule holds in LHM_d, then it also holds both in each later $LHM_{d'}$ ($d' > d$) and in $K\uparrow$. Note that the head of an extended clause is a disjunction of existentially quantified conjunctions of atoms, which constitutes

the main difference with Proposition 3.4.1. A second result relates the truth of an instance of an open formula to the truth of the normal instance of the same formula. A third result indicates a relationship between the fairness condition and the E-unification in NMGE: the body of an instance of a rule is true wrt LHM_d iff the instance is obtained by applying an E-unifier of the body of the rule with facts of M_d.

Proposition 3.4.2 *(a) Let I_1, I_2 be Herbrand interpretations of $\mathcal{L}$ and of an extension $\mathcal{L}'$ of $\mathcal{L}$ resp., such that $I_1 \subseteq I_2$. Let F be a closed formula based on $\mathcal{L}$ without negation and universal quantifiers. If $I_1 \models F$ then $I_2 \models F$.*

(b) Let F be an open formula and θ a grounding substitution of F. $LHM_d \models \theta(F)$ if and only if $LHM_d \models \gamma_d(\theta)(F)$. Here $\gamma_d(\theta)$ denotes the normalisation of θ wrt γ_d.

(c) Let $\theta(C)$ be a ground instance of a rule based on $\mathcal{L}+Sk_d$. θ is an E-unifier of the body of C and atoms in M_d wrt γ_d iff the body of $\theta(C)$ holds in LHM_d.

Proof Items (a) and (b) can easily be proved by induction on the structure of F. Intuitively, what goes wrong with negation and universal quantifiers in (a) is that a new fact in $I_2 \backslash I_1$ may contradict a negative fact in I_1 and a new domain element in $HU(\mathcal{L}') \backslash HU(\mathcal{L})$ may delete a universal property of I_1.

Item (c) clarifies the role of E-unification appearing in the definition of NMGE. Assume that the body of C is of the form $G_1, \ldots, G_k$. If θ is an E-unifier of G_j and some B in M_d wrt γ_d, then for each pair of arguments (t_i, s_i) of $\theta(G_j)$ and B, $EQ(\mathcal{L}) + \gamma \models t_i = s_i$. By the substitution axioms, $\theta(G_j)$ belongs to LHM_d.

Vice versa, assume $\theta(G_j)$ belongs to LHM_d. By Lemma 3.4.1 (c), there exists a B in M_d such that $\gamma_d(\theta(G_j)) = B$. Hence for each pair of terms (t_i, s_i) of $\theta(G_j)$ and B, $\gamma_d(t_i) \equiv s_i$. From this it follows that θ is an E-unifier of G_j and B, wrt γ_d. □

Theorem 3.4.1 (Soundness) *If K is a fair successful NMGE, then $K\uparrow_{\mathcal{L}}$ is a model for $<\mathcal{L}, P + E>$ and $P + E$ is consistent (a fortiori).*

We say that $K\uparrow_{\mathcal{L}}$ is the model generated by K.

Recall from definition 2.1.3 that an interpretation I' is called a symbol extension of an interpretation I iff the restriction of I' to the symbols of I is identical to I. The proof of the theorem and other proofs below use the following observation:

Lemma 3.4.2 *Let $\mathcal{L}'$ be an extension of $\mathcal{L}$, T a theory based on $\mathcal{L}$, M' an interpretation of $\mathcal{L}'$, and M the restriction of M' to the symbols of $\mathcal{L}$. Equivalently, M' is a symbol extension of M.*

Then M' is a model of $<\mathcal{L}', T>$ iff M is a model of $<\mathcal{L}, T>$.

The proof of this lemma is trivial, since the validity of a formula wrt an interpretation depends only on the interpretation of the symbols in the formula.

Proof (of Theorem 3.4.1)

First we prove that $K\uparrow_{\mathcal{L}}$ is a model of $<\mathcal{L}, E>$. Because of Lemma 3.4.2, it suffices to show that $K\uparrow$ is a model of $<\mathcal{L} + Sk(K), E>$. Consider the two sequences:

$$(A_d)_0^n = (LHM_d)_0^n = (LHM(<\mathcal{L} + Sk_d, EQ(\mathcal{L}) + M_d + \gamma_d>))_0^n$$
$$(B_d)_0^n = (LHM(<\mathcal{L} + Sk(K), EQ(\mathcal{L}) + M_d + \gamma_d>))_0^n$$

Both sequences consist of subsets of the Herbrand Base $HB(\mathcal{L}+Sk(K))$. We show that they have the same union, namely $K\uparrow$. One direction follows easily from the fact that for each d, $A_d \subseteq B_d$, therefore $K\uparrow = \bigcup_{d=0}^{n} A_d \subseteq \bigcup_{d=0}^{n} B_d$.

For the other direction, we must show that for each d:

$$LHM(<\mathcal{L} + Sk(K), EQ(\mathcal{L}) + M_d + \gamma_d>) \subseteq K\uparrow$$

Let A be an atom of $LHM(<\mathcal{L} + Sk(K), EQ(\mathcal{L}) + M_d + \gamma_d>)$. Because A contains only a finite number of skolem constants, there must be a $d' \geq d$ such that $A \in HU(\mathcal{L} + Sk_{d'})$. Because $E + M_{d'} + \gamma_{d'} \models M_d + \gamma_d$ and by Lemma 3.4.1(b), it holds that $EQ(\mathcal{L}) + M_{d'} + \gamma_{d'} \models M_d + \gamma_d$. As a consequence, A occurs in $LHM_{d'}$ and hence in $K\uparrow$.

$(B_d)_0^n$ is a monotonically increasing sequence of Herbrand models of the theory $<\mathcal{L} + Sk(K), E>$: from Lemma 3.4.1(b) follows that it is a sequence of models of $<\mathcal{L} + Sk(K), E>$ and that it is increasing can be proven in a similar way as proposition 3.4.1. A well-known property of clausal theories is that the fixpoint of a monotonically increasing sequence of models is a model. Since E is a clausal theory (by definition of *equality theory with completion*), $K\uparrow$ is a model of $<\mathcal{L} + Sk(K), E>$.

It remains to be proved that $K\uparrow$ is a model of P. Assume that there exists a ground instance $G_1, \ldots, G_k \rightarrow E_1, \ldots, E_l$ of a rule of P which is not satisfied by $K\uparrow$. So none of $E_1, \ldots, E_l$ holds in $K\uparrow$, and $G_1, \ldots, G_k$ hold in $K\uparrow$. However, since $(LHM_d)_0^n$ is monotonic, there exists a d such that $G_1, \ldots, G_k$ is in LHM_d. Since K is fair, there is a d' such that at least one E_j holds in $LHM_{d'}$. By Proposition 3.4.2(a), E_j also holds $K\uparrow$. This is in contradiction with our assumption.

□

To state the completeness result, we require an additional concept: the NMGE-Tree. Analogously with the concept of SLD-Tree, an NMGE-Tree is a tree of NMGEs obtained by applying all different conclusions of one rule in the descendants of a node.

Definition 3.4.5 *Let $\mathcal{L}$ be a language, E an equality theory with completion, P an extended program based on $\mathcal{L}$.*

An NMGE-Tree (NMGET) W for $<\mathcal{L}, P+E>$ is a tree such that:

- *Each node is labelled with a tuple (Sk, M, γ) where Sk is a skolem set, M a set of non-equality facts based on $\mathcal{L}+Sk$, and γ is a ground TRS based on $\mathcal{L}+Sk$.*
- *To each non-leaf N, a ground instance Cl of a rule of P is associated. For each conclusion with index j in the head of Cl, there is an arc leaving from N which is labelled by (Cl, j). If Cl has no conclusion then one arc leaves with label Cl.*
- *The sequence of labels on the nodes and arcs on each branch of W constitute an NMGE.*

Definition 3.4.6 *An NMGET is fair if each branch is fair.*

Definition 3.4.7 *An NMGET is failed if each branch is failed.*

Observe that a failed NMGET contains only a finite number of nodes. Also if T is inconsistent then because of the soundness Theorem 3.4.1, each fair NMGET is failed.

As a completeness result, we want to state that for any model of $P+E$, the NMGE contains a branch generating a *smaller* model. In a context of Herbrand models, the smaller-than relation can be expressed by set inclusion. However, because of the existential quantifiers and the resulting skolem constants, we cannot restrict to Herbrand models only. In order to define a smaller-than relation for general models, we must have a mechanism to compare models with a different domain. A solution to this problem is provided by the concept of *homomorphism.* Recall that an interpretation I is defined as a triple of a domain, pre-interpretation of the functor symbols and a truth function $\mathcal{H}_I$ mapping simple facts on truth values. Here, interpretations are two-valued. Truth values are ordered according to $\mathbf{f} < \mathbf{t}$.

Definition 3.4.8 *Let I_1, I_2 be interpretations of a language $\mathcal{L}$ with domains D_1, D_2.*

A homomorphism from I_1 to I_2 is a mapping h: $D_1 \rightarrow D_2$ which satisfies the following conditions:

- *For each functor f/n $(n \geq 0)$ of $\mathcal{L}$ and $x, x_1, \ldots, x_n \in D_1$:*

$$x \equiv I_1(f(x_1, \ldots, x_n)) \Rightarrow h(x) \equiv I_2(f(h(x_1), \ldots, h(x_n)))$$

- *For each predicate symbol p/n $(n \geq 0)$ of $\mathcal{L}$ and $x_1, \ldots, x_n \in D_1$:* $\mathcal{H}_{I_1}(p(x_1, \ldots, x_n)) \leq \mathcal{H}_{I_2}(p(h(x_1), \ldots, h(x_n)))$

Intuitively a homomorphism is a mapping from one domain to another, such that all positive information in the first model is maintained under the mapping. Therefore the homomorphisms in the class of models of a theory can be used to represent a "...*contains less positive information than...*" relation. We denote the fact that there exists a homomorphism from interpretation I_1 to I_2 by $I_1 \preceq I_2$. This notation captures the intuition that I_1 contains less positive information than I_2.

For NMGETs we can prove the following powerful completeness result.

Theorem 3.4.2 (Completeness) *Let E be an equality theory with completion, P an extended program, both based on $\mathcal{L}$.*

1. *There exists a fair, non-redundant NMGET for $<\mathcal{L}, P + E>$.*
2. *For each model M of $<\mathcal{L}, P + E>$ and each fair NMGET W, there exists a successful branch K of W such that $K\uparrow_{\mathcal{L}} \preceq M$.*

The first item in the proof is concerned with the fairness condition. The condition of fairness is quite strong and is stated in a non-constructive way. The proof provides a construction of a fair NMGE. This construction is based on the following two lemmas. The first lemma states that after applying a rule, the rule holds. The second lemma constructs a sequence which contains each ground instance of each rule an infinite number of times. This sequence will be used to construct a fair NMGE. Starting from the first clause, for each element in the sequence it is tested on whether or not the rule is violated (not satisfied). A violated rule is applied. In this way, we obtain a fair NMGE, since each rule is tested an infinite number of times and is applied when violated.

Lemma 3.4.3 *If a ground instance $\theta(C)$ of a rule C is applied at step d, then the conclusion of $\theta(C)$ holds in LHM_d.*

Proof The straightforward proof is omitted. □

Lemma 3.4.4 *Let P be an extended program based on $\mathcal{L}$, $\mathcal{L}_{sk}$ a countable alphabet of skolem constants. There exists a countable sequence (C_g) which contains every ground instance of a rule based on $\mathcal{L} + \mathcal{L}_{sk}$ an infinite number of times.*

Proof A well-known result is that for any countable set, the set of finite sequences of this set is countable. Each ground instance of a rule is a finite sequence of the countable set of functor symbols of $\mathcal{L}$ and $\mathcal{L}_{sk}$, logical connectors, logical quantifiers, brackets and the period ",". Therefore the set of possible ground instances of rules is countable.

So there is a countable sequence (C'_g) which contains each ground instance of a rule of P, based on $\mathcal{L} + \mathcal{L}_{sk}$ at least one time. This sequence can easily be transformed to the desired sequence:

$$(C_g)_0^\infty = (C'_1, C'_1, C'_2, C'_1, C'_2, C'_3, C'_1, \ldots)$$

or more formally: $(C_g)_0^\infty$ is obtained by concatenating the finite sequences $(C'_1, \ldots, C'_n)$ for increasing n. □

Proof (of Theorem 3.4.2)

First, using the sequence (C_g) from Lemma 3.4.4, we can construct a fair NMGET. With each node N, starting with the root, an index $g(N)$ is associated which points to the rule in (C_g) whose normalization is applied to obtain N. The descendants of N are obtained by searching the first rule C_h in (C_g), such that C_h is violated and $h > g(N)$. It is the normalisation of this rule which is applied to obtain the descendants of N, and for each descendant N' of N, $g(N')$ is defined as h. Technically the construction proceeds by induction on the depth of the nodes:

- the root N_0 is defined as in the definition of NMGET. We define $g(N_0) = -1$.
- let N_d be a non-failed node on depth d with index $g(N_d)$. N_d is a leaf of a branch $(N_0, \ldots, N_d)$, with associated sequences $(Sk_i)_0^d$, $(M_i)_0^d$, $(\gamma_i)_0^d$, $(LHM_i)_0^d$ and $(Cl_i, j_i)_1^d$.
  Now we look for the first rule C_h in (C_g), such that $h > g(N_d)$, C_h is based on $\mathcal{L}{+}Sk_d$ and C_h is violated in LHM_d. If such a C_h does not exist anymore, then we obtain a finite fair branch and N_d is a leaf in the constructed tree. If C_h is found, then it is of the form $\theta(C)$ where C is a rule of P. Let θ' be $\gamma_d(\theta)$. Since $\theta(C)$ has a true body, $\theta'(C)$ also has a true body wrt LHM_d (Proposition 3.4.2(b)). By Proposition 3.4.2(c), θ' is a normal E-unifier wrt γ_d of the atoms in the body of C and facts of M_d. So $\theta'(C)$ can be selected. The descendants of N_d are obtained by applying $\theta'(C)$ with each conclusion. For each descendant N, we define $g(N) = h$.

It is easy to see that this NMGET is non-redundant: only violated rules are applied. The NMGET is fair: if some rule Cl based on the language of some node N is violated in LHM_N, this rule reappears in the sequence $(C_g)_{g(N)+1}^\infty$ at least one time, say as the h'th element $(h > g(N))$. Because g strictly increases for descendants, in each non-failing branch departing from N, the integrity of C_h will be restored after at most $h - g(N)$ steps: either "by accident" by applying other rules of $C_{g(N)+1}, \ldots, C_{h-1}$, or by applying the normalisation of C_h (Lemma 3.4.3, Proposition 3.4.2(b)).

Now we prove the second part of the completeness theorem. The idea of the proof is as follows. We will construct by induction a path K through the NMGET W, such that for each node N_d on the path, LHM_{N_d} can be mapped into M by a homomorphism. At the $d+1$'th step, the selected rule has a true body in LHM_{N_d} and hence in M. Therefore, one of the conclusions of the selected rule must hold in M. We extend the path by selecting

the descendant of N_d corresponding to this conclusion. As a consequence, the homomorphism from LHM_{N_d} to M can be extended to $LHM_{N_{d+1}}$. The resulting branch K returns a fair NMGE such that $K\uparrow_{\mathcal{L}} \preceq M$.

Using W and M, we construct a branch $K = (N_d)_0^n$ in W with corresponding NMGE $(Sk_d)_0^n$, $(M_d)_0^n$, $(\gamma_d)_0^n$, and $(Cl_d, j_d)_1^n$ and a sequence of interpretations $(I_d)_0^n$ of $\mathcal{L}+Sk_d$, such that for each d the following invariant relation holds:

(a) I_d is a symbol extension of M by assigning to each skolem in Sk_d an element of the domain of M.

(b) If $d > 0$, then I_d is a symbol extension of I_{d-1} by assigning to each skolem in $Sk_d \backslash Sk_{d-1}$ an element of the domain of M.

(c) I_d is a model of $<\mathcal{L} + Sk_d, E + P>$.

(d) I_d is a model of $<\mathcal{L} + Sk_d, E + M_d + \gamma_d>$.

Note that the extension $\tilde{I}_d$ of I_d to the Herbrand universe $HU(\mathcal{L} + Sk_{N_d})$ defines a mapping from LHM_{N_d} to the domain of M and I_d. As we will show, this mapping is a homomorphism from LHM_{N_d} to M.

The branch is found by induction on the depth d:

- N_0 is the root of W. I_0 is defined as M. Clearly the invariant relation holds.
- Assume that we have found a path in W of length $d-1$, and N_{d-1} is not a leaf of W. By definition of NMGET, the descendants of N_{d-1} are obtained by applying all the conclusions of the same ground instance Cl_d of a rule C of P:

  $$Cl_d = G_1, \ldots, G_k \rightarrow E_1, \ldots, E_l$$

  By Proposition 3.4.2(c), the $G_1, \ldots, G_k$ hold in the least Herbrand model LHM_{d-1} of $<\mathcal{L} + Sk_{d-1}, E + M_{d-1} + \gamma_{d-1}>$. By the invariant, I_{d-1} is a model of $<\mathcal{L} + Sk_{d-1}, E + M_{d-1} + \gamma_{d-1}>$, therefore $G_1, \ldots, G_k$ hold wrt I_{d-1}. Since I_{d-1} is a model of P, it is a model of the rule C of which Cl_d is an instance. Therefore, for at least one i, E_i holds wrt I_{d-1}. We select the i'th descendant of N_{d-1} as N_d.

  Now we extend I_{d-1} for the new symbols of $Sk_d \setminus Sk_{d-1}$. Let E_i be $\exists Y_1, \ldots, Y_m : s_1 = t_1 \wedge \ldots \wedge s_g = t_g \wedge F_1 \wedge \ldots \wedge F_h$. Let V be the variable assignment $\{Y_1/a_1, \ldots, Y_m/a_m\}$ such that $V(s_1 = t_1 \wedge \ldots \wedge s_g = t_g \wedge F_1 \wedge \ldots \wedge F_h)$ holds wrt I_{d-1}. Let each Y_j be assigned the skolem constant sk_j in N_d. We extend I_{d-1} to I_d by defining $I_d(sk_j) = a_j$.

Clearly (a) and (b) of the invariant relation hold. (c) is a direct consequence of (a), the fact that I_d is a symbol extension of M and Lemma 3.4.2. That (d)

holds can be seen as follows. I_d is a symbolic extension of I_{d-1}, therefore I_d is a model of $<\mathcal{L} + Sk_d, E + M_{d-1} + \gamma_{d-1}>$ (Lemma 3.4.2). By construction of I_d, I_d is also a model of $\{s_1 = t_1, \ldots, s_g = t_g\}$ and of $\{F_1, \ldots, F_h\}$. One easily verifies that the following proposition holds:

$$E \models (M_{d-1} + \{F_1, \ldots, F_h\} + \gamma_{d-1} + \{s_1 = t_1, \ldots, s_g = t_g\} \Leftrightarrow M_d + \gamma_d)$$

Since I_d is a model of E, I_d is a model of $M_d + \gamma_d$.

The construction above returns a successful branch K in W. Since W is fair, K is a fair NMGE. Because of Theorem 3.4.1, $K\uparrow_{\mathcal{L}}$ is a model of $<\mathcal{L}, P + E>$. It remains to show that there is a homomorphism from $K\uparrow_{\mathcal{L}}$ to M. Because each I_d is consistent with all its predecessors in the sequence, the union of the sequence $(I_d)_0^n$ defines an interpretation I of $\mathcal{L} + Sk(K)$. Remember from section 2.1 how I is extended to a mapping $\tilde{I}$ from $HU(\mathcal{L} + Sk(K))$ to the domain of M. We prove that $\tilde{I}$ is a homomorphism from $K\uparrow_{\mathcal{L}}$ to M.

By its definition, $\tilde{I}$ trivially satisfies the first condition of *homomorphism*. The second condition of homomorphism is that for any atom $p(t_1, \ldots, t_n) \in K\uparrow$, it should hold that $M \models p(\tilde{I}(t_1), \ldots, \tilde{I}(t_n))$. Assume $p(t_1, \ldots, t_n) \in K\uparrow$. There exists a d such that $p(t_1, \ldots, t_n) \in LHM_d$. Hence, $E + M_d + \gamma_d \models p(t_1, \ldots, t_n)$ and because of invariant (d), $I_d \models p(t_1, \ldots, t_n)$. Since I is a symbol extension of I_d, $I \models p(t_1, \ldots, t_n)$. This is equivalent with $I \models p(\tilde{I}(t_1), \ldots, \tilde{I}(t_n))$ (by definition of truth of an atom wrt to some interpretation). Since I and M interpret p/n by the same relation, it holds that $M \models p(\tilde{I}(t_1), \ldots, \tilde{I}(t_n))$.

□

The construction of the fair NMGET in Theorem 3.4.2 does still not give a clue on how to implement the fairness condition in a practical way. This problem will be dealt with properly in section 3.6.

As a corollary we obtain the following reformulation of a traditional completeness result.

Corollary 3.4.2 *If $<\mathcal{L}, P + E>$ is consistent then in each fair NMGET there exists a successful branch.*

If there exists a failed NMGET for $<\mathcal{L}, P + E>$, then $<\mathcal{L}, P + E>$ is inconsistent, and all fair NMGETs are failed.

The completeness result does not imply that all models are generated. For example for $P = \{p \leftarrow q\}$, the model $\{p, q\}$ is not generated by an NMGE. The following example shows that different NMGETs for the same theory might generate different models.

Example $P = \{\ p, q\leftarrow \quad p\leftarrow\ \}$

Depending on which of these clauses is applied first, we get two different non-redundant NMGETs. If $p\leftarrow$ is applied first, then $p, q\leftarrow$ holds already and is not applied anymore. So we get an NMGET with one branch of length 1. On the other hand if $p, q\leftarrow$ was selected first, then two branches exist and we get the solutions $\{p\}$ and $\{p, q\}$.

Therefore it would be interesting if we could characterize a class of models which are generated by each NMGET. The second item of the completeness Theorem 3.4.2 gives some indication: for any given model M, some successful branch of the NMGET generates a model with less positive information than M. For the clausal case, models with no redundant positive information are minimal Herbrand Models. From this observation one would expect that for a clausal program, each fair NMGET generates all minimal models. Indeed, the following completeness theorem holds:

Theorem 3.4.3 (Minimal Herbrand models) *If P is clausal, then for each fair NMGET W, each minimal Herbrand model is generated by a branch in W.*

The proof is easy: for a clausal theory, each successful branch in each fair NMGET W generates a Herbrand model (since no skolemisation is necessary). From Theorem 3.4.2 it follows that for each minimal Herbrand model M, there exists a branch K in W such that $K\uparrow \preceq M$. Since for Herbrand models, $\preceq$ corresponds to $\subseteq$, and since M is minimal it follows that $K\uparrow = M$.

Now we return to the general case. Since we have to deal with non-Herbrand models, the concept of *minimal model* must be extended.

Definition 3.4.9 *Let T be a theory based on a language $\mathcal{L}$.*

A model M_1 has the same information content as *or* is IC-equivalent with *a model M_2 if there exists a homomorphism from M_1 to M_2 and a homomorphism from M_2 to M_1.*

A model M_m of $<\mathcal{L}, P>$ is minimal *iff for each model M' of T such that $M' \preceq M_m$, M' is IC-equivalent with M_m.*

The notion of IC-equivalence of models is weaker than the notion of isomorphism. Consider $T = \{p(a)\leftarrow \qquad \exists X : P(X)\leftarrow\}$. There exist two different non-redundant NMGE-trees, depending on the selection of the first extended clause. Selecting first the rule $p(a)\leftarrow$ one obtains $M_1 = \{\text{p(a)}\}$. Selecting the other rule first yields $M_2 = \{\text{p(sk), p(a)}\}$. These models are not isomorphic but they are IC-equivalent: $h_1(a)\equiv a$ defines a homomorphism from M_1 to M_2 and $h_2(a)\equiv h_2(sk)\equiv a$ defines a homomorphism from M_2 to M_1.

It is straightforward that *IC-equivalence* defines an equivalence relationship between models and that a model IC-equivalent with a minimal model is minimal

also. This definition is a generalization of the concept of minimality for Herbrand models: for clausal theories, one can easily prove using Theorem 3.4.3 that a model is minimal iff it is IC-equivalent with a minimal Herbrand model.

Each fair NMGET generates all minimal models, modulo IC-equivalence, but not modulo isomorphism:

Theorem 3.4.4 (Completeness on minimal models) *Let C be a class of IC-equivalent minimal models and W a fair NMGET. Then there exists a branch of W generating a minimal model of C.*

This theorem is a simple consequence of the completeness Theorem 3.4.2.

In [Bry90], the following completeness theorem for Satchmo-1 was formulated: Satchmo-1 is complete for *finite satisfiability*, i.e. if a theory T has a finite model, then Satchmo-1 generates a finite model. NMGE does not satisfy this property. Consider for example the following theory:

$$\{\exists X : p(a, X)) \leftarrow \qquad (\exists Z : p(Y, Z)) \leftarrow p(X, Y)\}$$

This theory has only an infinite NMGE, generating the model $\{p(a, sk_1)$, $p(sk_1, sk_2), p(sk_2, sk_3), \ldots\}$. However, the interpretation $\{p(a, a)\}$ is a finite model. Satchmo-1 generates both the finite an the infinite model.

This distinction between Satchmo-1 and NMGE is caused by a distinct treatment of existential quantifiers. In an NMGE, each existential variable is skolemised. On the other hand, Satchmo-1 keeps track of the domain of interpretation, and assigns each of the existing domain elements to the existential variable (giving rise to different branches in the computation) before introducing a new skolem constant as a final alternative. Hence, each Satchmo-1 computation tree comprises an NMGET (if no equality atoms in the head occur). As a consequence, the treatment of existential variables as in NMGE is more efficient for showing inconsistency of a theory whereas the treatment in Satchmo-1 is more suitable for showing consistency of a theory. However, it should be noted that the main issue of this chapter, i.e. the technique for dealing with equality, stands orthogonal on the way the existential quantifiers are dealt with. The techniques that are proposed here can as well be incorporated in a procedure which treats existential quantifiers as in Satchmo-1.

3.5 Duality of SLD+Abduction and Model Generation.

The NMGE framework allows to formalise the observations that were made in the introduction. We prove that FEQ is an equality theory with completion and that the completion procedure is dual to the unification procedure. We first introduce the notion of a dualisation more formally.

Definition 3.5.1 *Let $\mathcal{L}$ be a first order language with variables $\mathcal{L}_V$ and $\mathcal{L}_{sk}$ an alphabet of skolem constants which do not occur in $\mathcal{L}$.*

A dualisation mapping is a one-to-one correspondence $D : \mathcal{L}_{sk} \rightarrow \mathcal{L}_V$.

The dualisation mapping D can be extended to a mapping from $HU(\mathcal{L}+\mathcal{L}_{sk}) \cup HB(\mathcal{L}+\mathcal{L}_{sk})$ to the set of terms based on $\mathcal{L}$ by induction on the depth of terms:

- *for each constant c of $\mathcal{L}$: $D(c) \equiv c$*
- *for each term or atom $f(t_1, \ldots, t_n)$:* $D(f(t_1, \ldots, t_n)) \equiv f(D(t_1), \ldots, D(t_n))$

D can be further extended to any formula or set of formulas. Under dualisation, a ground TRS γ based on $\mathcal{L}+\mathcal{L}_{sk}$ corresponds to an equation set $D(\gamma)$ with terms based on $\mathcal{L}$.

Definition 3.5.2 *A ground TRS γ is said to be in solved form iff $D(\gamma)$ is an equation set in solved form [MM82].*

An equation set is in solved form iff it consists of equations $X_i = t_i$, such that the X_i's are distinct variables and do not occur in the right side of any equation. So a TRS is in solved form if the left terms are distinct skolem constants of $\mathcal{L}_{sk}$ which do not occur at the right. A TRS in solved form can also be seen as the dual of an idempotent variable substitution.

Proposition 3.5.1 *Let γ be a TRS in solved form.*

(a) γ is complete wrt to $<\mathcal{L}, \mathrm{FEQ}>$.

(b) For each compound term $f(t_1, \ldots, t_n)$, it holds that:

$$\gamma(f(t_1, \ldots, t_n)) \equiv f(\gamma(t_1), \ldots, \gamma(t_n))$$

and for each constant c of $\mathcal{L}$: $\gamma(c) \equiv c$.

Proof Item (b) follows straightforwardly from the fact that a TRS in solved form does not contain compound terms at the left.

With respect to item (a), it is intuitively clear that for each term t, its reduction tree is finite and all leaves contain the same term. Otherwise said, γ is noetherian and Church-Rosser. For a formal proof, we can rely on a theorem in Term Rewriting for *irreducible* TRSs. A TRS γ is called *irreducible* iff for each $s \rightarrow t \in \gamma$, t is in normal form wrt to γ and s is in normal form wrt $\gamma \backslash \{s \rightarrow t\}$. Clearly a TRS in solved form is irreducible. In [Met83] it was proved that an irreducible TRS is noetherian and Church-Rosser. By Proposition 3.3.2(a), a TRS in solved form is complete wrt $EQ(\mathcal{L})$.

It remains to be proved that for each pair of terms s, t based on an extension $\mathcal{L}'$ of $\mathcal{L}$ by constants, it holds that

$$s = t \in LHM(<\mathcal{L}', FEQ(\mathcal{L}) + \gamma>) \text{ iff } \gamma(s) \equiv \gamma(t)$$

Define $S = \{s = t | s, t \in HU(\mathcal{L}')$ and $\gamma(s) \equiv \gamma(t)\}$. Since γ is complete wrt $EQ(\mathcal{L})$, S is $LHM(<\mathcal{L}', EQ(\mathcal{L}) + \gamma>)$. Since $FEQ(\mathcal{L})$ is an extension of $EQ(\mathcal{L})$, it holds that $S \subseteq LHM(<\mathcal{L}', FEQ(\mathcal{L}) + \gamma>)$.

Vice versa, to prove is that S is a Herbrand model of $FEQ(\mathcal{L}) + \gamma$. It suffices to show that S satisfies the axioms in $FEQ(\mathcal{L}) \backslash EQ(\mathcal{L})$:

- Assume $\gamma(f(t_1, \ldots, t_n)) \equiv \gamma(f(s_1, \ldots, s_n))$. By item (b) of the proposition, it directly follows that for each i, $\gamma(t_i) \equiv \gamma(s_i)$. Hence $t_i = s_i$ belongs to S.
- Because of item (b), two terms with distinct main functors $f/$n and g/m cannot be rewritten to the same term by γ.
- Finally the consistency of the *occur-check* axioms must be proved. Assume that there exist a pair of terms s, t such that $\gamma(s) \equiv \gamma(t)$ and s contains t. However, again because of item (b), it holds that if s contains t then $\gamma(s)$ contains $\gamma(t)$. Hence, $\gamma(s)$ contains itself. This is impossible.

□

The theorem below expresses the procedural duality between the unification and completion, as announced in section 3.1. Here, the notion of procedural duality refers to a form of isomorphism between two procedures. Both procedure must be decomposable as sequences of basic operations. The isomorphism then refers to the fact that, if the procedures are activated on dual input, then there must be a one-to-one mapping between the two resulting sequences of basic operations, such that the input and output of each two corresponding operators are dual. In the theorem, we take unification as the first procedure with an equality set as input. The dual of the input is the ground TRS obtained by interpreting variables as skolem constants, and the dual of unification is completion.

Theorem 3.5.1 (Duality of completion and unification) FEQ($\mathcal{L}$) *is an equality theory with completion. The completion procedure is dual to unification. The dual of the completion of a ground TRS* γ *based on an extension* $\mathcal{L}'$ *of* $\mathcal{L}$ *with constants, is the mgu of* $D(\gamma)$. *Or* $D(TRS\text{-}comp(\gamma)) = mgu(D(\gamma))$.

Proof Below the algorithm of [MM82] is dually reformulated. The symbol x denotes a skolem constant, t a term, E denotes a set of equality atoms. The algorithm proceeds by iteratively transforming a TRS γ by applying the following rewrite rules:

(1) $\{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\} \cup E$
$\Rightarrow \quad \{t_1 = s_1, \ldots, t_n = s_n\} \cup E$

(2) $\{f(t_1, \ldots, t_m) = g(s_1, \ldots, s_n)\} \cup E$ where $f/m \not\equiv g/n$

$\Rightarrow \quad \{\Box\}$

(3) $\{x = x\} \cup E \quad \Rightarrow \quad E$

(4) $\{t = x\} \cup E$ where t is not a skolem constant
$\Rightarrow \quad \{x = t\} \cup E$

(5) $\{x = t\} \cup E$ where $x \not\equiv t$ and x appears in t
$\Rightarrow \quad \{\Box\}$

(6) $\{x = t\} \cup E$ where $x \not\equiv t$, x does not appear in t, and x has another occurrence in E
$\Rightarrow \quad \{x = t\} \cup \{x/t\}(E)$

The algorithm terminates when no rewrite rule can be applied. Its termination follows directly from the termination of the dual algorithm ([MM82]). It returns $\{\Box\}$ or a TRS δ in solved form. By Proposition 3.5.1, it follows that δ is complete. To see that γ and δ are equivalent wrt $FEQ(\mathcal{L})$, just verify that each rewrite rule maintains equivalence wrt $FEQ(\mathcal{L})$.

□

We call δ the solved form of γ.

An interesting property of the completion wrt FEQ is that it is *incremental.*

Definition 3.5.3 *The composition operation "o" on Term Rewriting Systems in solved form is defined as the dual of composition of variable substitutions. Or, let γ_1, γ_2 be complete Term Rewriting Systems, dual to the substitutions θ_1, θ_2. Then $\gamma_1 o \gamma_2$ is defined as $D^{-1}(\theta_1 o \theta_2)$.*

Proposition 3.5.2 *Let γ, δ be two ground TRSs, γ_c the completion of γ, δ_c the completion of $\gamma_c(\delta)$. Then, $\delta_c o \gamma_c$ is a completion of $\gamma \cup \delta$.*

Proof Applying the completion algorithm on $\gamma \cup \delta$, it is always possible to first transform γ into solved form, thus obtaining γ_c. One easily verifies that during this transformation, δ is transformed gradually to $\gamma_c(\delta)$, the normalized form wrt γ_c. So, the result of this first stage is $\gamma_c \cup \gamma_c(\delta)$. Then the completion proceeds by bringing $\gamma_c(\delta)$ in solved form, which returns δ_c. Similarly to the first phase, the effect on the equations of γ_c is that all terms are normalized wrt to δ_c. So, the total equation set is $\delta_c(\gamma_c) \cup \delta_c$. This is nothing else than $\delta_c o \gamma_c$.

Notice that this result is dual to the property of unification that if θ is a mgu of an equation set E_1 and σ is the mgu of $\theta(E_2)$, then $\theta o \sigma$ is the mgu of $E_1 \cup E_2$. □

Also interesting from a practical point of view is that E-unification wrt a TRS in solved form collapses to unification:

Proposition 3.5.3 *Let γ be a TRS in solved form, s,t normal terms. θ is a normal E-unifier of (s,t) wrt γ iff θ is a unifier of s and t.*

Proof Assume θ is a normal E-unifier of (s,t). Hence, $EQ(\mathcal{L}) + \gamma \models \forall(\theta(s) = \theta(t))$ and since γ is complete, $\gamma(\theta(s))\equiv\gamma(\theta(t))$. Since s,t and θ are normal, none of the skolem constants at the left of γ appear in them. Hence $\theta(s)\equiv\gamma(\theta(s))\equiv\gamma(\theta(t))\equiv\theta(t)$. So θ is a unifier of s and t.

Vice versa, any unifier is a trivial E-unifier. □

A direct consequence of this proposition is that step (a) in the NMGE process can be simplified by replacing E-unification by unification. Indeed, the terms occurring in M_{d-1} are in normal form wrt γ_{d-1}. Also, all terms in the body of a rule of P are in normal form wrt γ_{d-1} since they are based on $\mathcal{L}$.

As was observed in the introduction, the duality between unification and completion can be extended further to the complete process of SLD+Abduction. The latter procedure is a simple extension of SLD-resolution for definite abductive programs [CP86]. Distinction is made between defined predicates which have a definition (i.e. a possibly empty set of definite clauses with head matching the predicate) and abductive predicates which have no definition. An SLD+Abduction refutation is a finite SLD-derivation during which only atoms of defined predicates are selected for resolution and such that the final resolvent contains only abductive atoms. So, given a definite abductive program P and a definite query Q, we can describe an SLD+Abduction derivation of length n, $n \in \mathbb{N} \cup \{\infty\}$, for Q as usual as a triplet of sequences:

- $(R_d)_1^n$ of resolvents with $R_1 = Q$.
- $(C_d)_2^n$ of renamings of program clauses, sharing no variables with each other or with Q.
- $(\theta_d)_2^n$ of substitutions

such that each R_d $(d > 1)$ is derived from R_{d-1} using C_d and θ_d.

Below we assume without loss of generality that no "="-atom occurs neither in a body of a definite clause of P or in Q. If this special predicate was to occur in P, rename it by a new predicate, for example by $eq/2$, whose definition consists of the unique clause:

$$eq(X,X)\leftarrow$$

The SLD+Abduction procedure takes as input a definite abductive program and definite query. Below we define the dual interpretation of the input. Recall that a query $Q = \leftarrow L_1,\ldots,L_n$ denotes a formula of the form $\forall(\neg L_1 \vee \ldots \vee \neg L_n)$. Therefore, $\neg Q$ denotes the formula $\exists(L_1 \wedge \ldots \wedge L_n)$.

Definition 3.5.4 *Given a definite abductive program P and a definite query Q. We define the dual P_D of (P, Q) as the extended program* only-if$(P) \cup \{\neg Q\}$.

An example of a pair of a program and query and its dual were given in figure 3.2.

Lemma 3.5.1 *$P_D \backslash$FEQ is a range restricted extended program. For each defined predicate p/n of P, one rule C_p occurs in P_D, having $p(\overline{X})$ as the unique atom in its body.*

Range restricted means that every universal variable which occurs in the head occurs in the body. The lemma is a straightforward consequence of the definition of *only-if*(P).

Below, the duality between SLD+Abduction and NMGE suggested by the example in section 3.1 is expressed formally. Informally, the selection of a defined atom $p(\bar{t})$ corresponds dually to the selection of the instance of the rule C_p, having the dual of the selected atom in its body. With each clause in the definition of p corresponds a conclusion in C_p. Therefore, we can associate with the selection of a clause the selection of a conclusion of the rule. The unification of the selected atom and the head of the clause corresponds dually to the completion operation of the equality atoms in the selected conclusion. The application of the mgu on resolvent corresponds to the normalisation.

Lemma 3.5.2 *Let $\mathcal{L}$ be a first order language, with an alphabet of variables $\mathcal{L}_V$, $\mathcal{L}_{sk}$ an alphabet of skolem constants disjunct from $\mathcal{L}$, and $D : \mathcal{L}_{sk} \rightarrow \mathcal{L}_V$ a dualisation mapping. Let P be a definite abductive program, Q a definite query, both based on $\mathcal{L}$. Let $(R_d)_1^n$, $(C_d)_2^n$ and $(\theta_d)_2^n$ be an SLD-derivation for (P, Q) and $p_d(t_1, \ldots, t_{n_d})$ the atom selected at step d.*

A unique rule C_{p_d/n_d} in P_D corresponds with p_d/n_d and a conclusion E_{j_d} corresponds with the clause C_d. Let σ_d be the unifier of the body of C_{p_d/n_d} with $D^{-1}(p_d(t_1, \ldots, t_{n_d}))$. We define Cl_d, Sk_d, γ_d and M_d as follows:

$$
\begin{array}{ll}
Cl_1 = \neg Q, & Cl_d = \sigma_d(C_{p_d/n_d}) \\
Sk_0 = \{\}, & Sk_d = D^{-1}(var(\{Q, C_2, C_3, \ldots, C_d\}) \\
\gamma_0 = \gamma_1 = \{\}, & \gamma_d = D^{-1}(\theta_d o \ldots o \theta_1) \\
M_0 = \{\}, & M_d = D^{-1}(\{\theta_d o \ldots o \theta_{i+1}(G_j) | 0 \leq i \leq d \text{ and} \\
 & G_j \text{ is an atom in } R_i\}.
\end{array}
$$

The tuple of sequences $(Sk_d)_0^n$, $(M_d)_0^n$, $(\gamma_d)_0^n$ and $(Cl_d, j_d)_1^n$ defines a NMGE.

As an example consider the successful SLD+Abduction derivation in figure 3.4. The sequence of resolvents is:

$$\leftarrow p(X, X) \quad \leftarrow q(a, V)$$

The sequence of mgus is:

$$\{\} \qquad \{X/a\}$$

The sequence $(Sk_d)_0^2$ of the dual NMGE is

$$\{\} \qquad \{sk_1\} \qquad \{sk_1, sk_2\}$$

where $D(sk_1) = X$ and $D(sk_2) = V$. The sequence $(\gamma_d)_0^2$ is

$$\{\} \qquad \{\} \qquad \{sk_1 \rightarrow a\}$$

The sequence of derived facts of the dual NMGE is

$$\{\} \qquad \{p(sk_1, sk_1)\}\{p(a,a), q(a, sk_2)\}$$

Proof The lemma can be checked by a straightforward case analysis of the operations that occur during a resolution step and a NMGE computation step. The following correspondences are easily shown.

The selection of the atom in the resolvent and the clause correspond dually to the selection of the rule and the conclusion respectively. Here we need the fact that each rule in P_D is range restricted: each universal variable in the rule occurs in the body. If that was not the case, then additional choices had to be made to instantiate the variables occurring in the conclusion only. The duality would be broken.

The renaming of the program clause can be seen as the dual of the skolemisation. This is because the used clauses and the query do not share variables.

The computation of the mgu corresponds to the completion of the set of equalities. This is due to the fact that the completion can be computed incrementally (Lemma 3.5.2) and the fact that the equation set to be solved for the unification corresponds exactly to the dual of the set of equality atoms to be completed.

The application of the mgu and the addition of the literals of the used clause to the resolvent correspond to the normalisation and assertion phase.

□

Theorem 3.5.2 *For any definite query Q, an abductive refutation for Q and P can be dually interpreted as a successful fair NMGE for* only-if(P) + $\neg Q$. *The set of atoms of the generated model, restricted to the abducible predicates is the dual of the abductive solution. The dual of the answer substitution is the restriction of γ_n to the skolem constants dual to the variables in the query.*

A failed SLD+Abduction derivation corresponds dually to a failed NMGE.

A fair SLD+Abduction derivation corresponds dually to a fair NMGE.

A (fair) SLD+Abduction tree corresponds dually to a (fair) NMGE-tree.

Proof An SLD+Abduction derivation is failed when in the last resolvent an atom of a defined predicate p/n with an empty definition is chosen or else a clause whose head does not unify with the atom. In the first case, the rule $C_{p/n}$ is of the form $p(X_1, \ldots, X_n) \rightarrow$. Hence the NMGE fails. In the second case, the completion of the atoms returns $\{\Box\}$ and the NMGE fails also.

In a fair SLD-derivation, each atom occurring in a resolvent is eventually selected. That this implies that the dual NMGE is fair seems evident. Formally, the proof goes as follows. Consider any defined atom $A = p(t_1, \ldots, t_n)$ in LHM_d and the corresponding definition

$$C_{p/n} = p(\overline{X}) \rightarrow E_1, \ldots, E_l$$

Define $\sigma_A = \{X_1/t_1, \ldots, X_n/t_n\}$. We show that $LHM_{d'} \models \sigma_A(E_1 \vee \ldots \vee E_l)$ for some $d' \geq d$. This is equivalent of showing $LHM_{d'} \models \gamma_{d'}(\sigma_A)(E_1 \vee \ldots \vee E_l)$ (by Proposition 3.4.2(b)).

Since A occurs in LHM_d, $\gamma_d(A) \in M_d$ (Lemma 3.4.1(c)). Two possibilities exist (by Lemma 3.5.2): or $\gamma_d(A)$ is the normalisation of an atom $B = \sigma_B(p(\overline{X}))$ selected for step d or earlier, or $\gamma_d(A)$ is the dual of an atom in R_d.

In the first case $LHM_d \models \gamma_d(\sigma_B)(E_1 \vee \ldots \vee E_l)$ (Lemma 3.4.3 and Proposition 3.4.2(a+b)). It suffices to prove that $\gamma_d(\sigma_A) \equiv \gamma_d(\sigma_B)$. We have that $\gamma_d(\sigma_A)(p(\overline{X}))) = \gamma_d(A) = \gamma_d(B) = \gamma_d(\sigma_B(p(\overline{X}))$. The identity of $\gamma_d(\sigma_A)$ and $\gamma_d(\sigma_B)$ follows straightforwardly from this equation and the fact that both substitutions have $\overline{X}$ as domain.

We find that $LHM_d \models \gamma_d(\sigma_A)(E_1 \vee \ldots \vee E_l)$ and we can take $d' = d$. The second case can be proven in an analogous way using the fairness of the SLD-derivation.

Since the selection in a refutation is fair, a refutation corresponds to a fair successful NMGE.

Because of all previous results, a fair SLD+Abduction tree corresponds to a fair NMGET.

□

What happens if we drop *FEQ* from *only-if*(P)? In that case, we must replace it by *EQ*. This implies that the completion procedure of *FEQ*, i.e. the dual interpretation of unification must be replaced by a completion procedure of *EQ*, for example Knuth-Bendix completion. As a consequence the declarative and procedural duality between the model generation and the abduction ceases to exist. Consider the following trivial program P and query Q:

$r(a) \leftarrow$
$\leftarrow r(b)$

SLD(+Abduction) will fail on this query and this corresponds dually to the fact that with FEQ, *only-if*$(P) + \neg Q$ is inconsistent. If we replace FEQ by EQ in *only-if*(P), the set $\{a = b, r(a)\}$ can be extended to a model of *only-if*$(P) +$ *not*(Q). This model corresponds to the abductive solution $\{a = b\}$. Most current abductive procedures will not return this solution and this shows that currently, FEQ is inherently present in most current work on abduction in LP.

The following corollary was proved first by Clark [Cla78] for normal programs. For the definite case it follows immediately from the theorem above.

Corollary 3.5.1 *An SLD-refutation for a query Q and a definite program P without abductive predicates is a consistency proof of* only-if$(P) + \neg Q$. *A failed SLD-tree for a ground query Q and P is an inconsistency proof of* only-if$(P) + \neg Q$, *and therefore of comp*$(P) + \neg Q$.

Theorem 3.5.2 gives a duality in one direction: an SLD+Abduction refutation can be dually interpreted as a fair NMGE. The reverse direction does not hold: there exists fair NMGEs which do not correspond to SLD+Abduction refutations and there exists models generated by fair NMGEs which do not correspond to abductive solutions. Here is a trivial example of this situation. Consider the definite program $P = \{p \leftarrow p\}$ and the query $\leftarrow p$ without undefined predicates. No SLD-refutation for the query exists and the SLD-tree consists of one infinite branch $p \leftarrow p \leftarrow p \leftarrow \ldots$. *only-if*$(P) + \neg Q$ is the theory $\{p \leftarrow \quad p \leftarrow p\}$. The dual of the infinite SLD-derivation is a fair NMGE and generates the model $\{p\}$. The NMGE dually corresponds to an SLD-derivation but not to an SLD-refutation. In general, in a fair SLD+Abduction tree, all branches correspond dually to fair NMGEs, and hence dually generate models. However, only the finite branches generate abductive solutions. So, in the case of an infinite branch, the duality is broken.

What this example shows is that infinite fair NMGEs may generate models which do not correspond to abductive solutions. Do finite NMGEs always generate models corresponding to abductive solutions? Unfortunately, this is not the case either. Consider the following NMGE for the same theory as in the previous paragraph: $M_0 = \phi; M_1 = \{p\}$. This is a finite fair NMGE, but the set of abductive atoms in the model $(= \phi)$ is not an abductive solution.

There is an important class of definite abductive programs where the duality is perfect, namely for definite abductive *acyclic programs* and *bounded queries* [AB90]. For these programs and queries, an SLD+Abduction tree is always finite. Using this fact and the completeness Theorem 3.4.2, it is easy to prove that the abductive atoms in each model of the dual theory form an abductive solution.

3.6 Implementing NMGE

We have implemented two instances of the NMGE framework, one for FEQ and one for EQ. Both instances are implemented not only for only-if parts of definite

logic programs, but more generally for theories consisting of extended clauses. The model generator for FEQ is easy to implement, since E-unification can be replaced by unification (Proposition 3.5.3) and FEQ has a simple, incremental completion procedure. Two technical problems deserve special attention. One problem is that all universal variables of a rule being applied must be instantiated with ground terms and the procedure matching bodies of rules with elements of M_d only instantiates variables of the body. We circumvent this problem by requiring that the rules are in *range restricted form* (i.e. all universal variables occurring in the head occur in the body in a non-equality atom), and by transforming each theory violating this condition to range restricted form. This can be done by introducing a domain predicate $\mathcal{U}/1$ representing the domain of interpretation. For each universal variable X not occurring in the body, $\mathcal{U}(X)$ is added to the body. For each existential variable X in a disjunct of the head, $\mathcal{U}(X)$ is added to the disjunct. For example, a rule

$$p(X,X), q(f(X)) \rightarrow \exists Z : p(g(X,Z),Y)$$

is transformed to:

$$p(X,X), q(f(X)), \mathcal{U}(Y) \rightarrow \exists Z : \mathcal{U}(Z) \wedge p(g(X,Z),Y)$$

In addition, rules of the form $\mathcal{U}(X_1),\ldots,\mathcal{U}(X_n) \rightarrow \mathcal{U}(f(X_1,\ldots,X_n))$ are added for each functor f/n $(n \geq 0)$.

A second problem is related to the fairness condition. Theorem 3.4.2 proves that a fair NMGET exists, but without clarifying how to implement the condition. The solution that we have adopted is the one used in Satchmo [MB87]: *level saturation*. The idea is to generate conclusions level by level. For a given level with associated M_d, γ_d, normal instances of rules which are violated in LHM_d are selected, conclusions are selected, skolemisation is performed but all facts in the selected conclusion are stored apart. Only when all violated instances have been applied, the completion γ_{d+1} of γ_d and the derived equality facts is computed, the derived non-equality facts are added to M_d and normalisation is applied, yielding M_{d+1}.

A second instance that was implemented is for EQ as underlying equality theory. The completion of a *ground* TRS can be computed [Der87], [Sny89] and moreover, efficient algorithms exist ([Sny89]). Hence, EQ is an equality theory with completion. Our prototype uses *narrowing* [MMR86] to compute normal E-unifiers, and an optimised form of the Knuth-Bendix algorithm [KB70] as completion procedure. The model generator operates on range restricted programs. The fairness condition is implemented using *level saturation*.

Experiments with both systems are promising. They show that the dynamic contraction, implemented by dynamic completion and normalisation, often avoids exponential explosion and looping caused by the equality axioms. However, at this point our prototypes are too primitive to be already of practical interest, as

was proven by the experiment described in the next section. In the future, the implementation should be refined.

3.7 Executing declarative specifications

In chapter 1, we argued that model generation is an interesting problem solving paradigm for declarative specifications. In principle, model generation is of use whenever the problem to be solved is to find a possible state of some domain, satisfying some set of constraints.

A type of problem which often can be naturally formulated in these terms, is a puzzle. We illustrate this with a solution of the well-known five-houses puzzle, also called the zebra puzzle. Five houses have different colours and are occupied by persons of different nationality. Each person has his own drink, pet and profession. Additional information about the houses and their attributes is given. We use the predicates $has\_nat/2$, $has\_color/2$ $has\_drink/2$, $has\_prof/2$, $has\_pet/2$ to represent the relations between the houses and their different attributes. The constraints are specified as follows:

The Englishman lives in the red house:
$\exists H : has\_nat(H, englishman) \wedge has\_color(H, red)$
The Spaniard has a dog:
$\exists H : has\_nat(H, spaniard) \wedge has\_pet(H, dog)$
The Japanese is a painter:
$\exists H : has\_nat(H, japanese) \wedge has\_prof(H, painter)$
The Italian drinks tea:
$\exists H : has\_nat(H, italian) \wedge has\_drink(H, tea))$
The Norwegian lives in the first house on the left:
$has\_nat(h1, norwegian)$
In the green house, one drinks coffee:
$\exists H : has\_drink(H, coffee) \wedge has\_color(H, green))$
The white house is next to the blue one:
$\exists H1, H2 : has\_color(H1, white) \wedge next(H1, H2) \wedge$
$has\_color(H2, green)$
The sculptor breeds snails:
$\exists H : has\_prof(H, sculptor) \wedge has\_pet(H, snails)$
In the yellow house lives a diplomat:
$\exists H : has\_prof(H, diplomat) \wedge has\_color(H, yellow)$
In the middle house, one drinks milk:
$has\_drink(h3, milk)$
The blue house is near the house of the Norwegian:
$\exists H1, H2 : has\_color(H1, blue) \wedge near(H1, H2) \wedge$
$has\_nat(H2, norwegian)$

The violinist drinks wine:
$\exists H : has\_drink(H, wine) \wedge has\_prof(H, violinist)$
The person with the fox lives near the doctor:
$\exists H1, H2 : has\_pet(H1, fox) \wedge near(H1, H2) \wedge$
$has\_prof(H2, doctor)$
The one with the horse lives near the diplomat:
$\exists H1, H2 : has\_pet(H1, horse) \wedge near(H1, H2) \wedge$
$has\_prof(H2, diplomat)$
Somebody has a zebra and somebody drinks water:
$\exists H : has\_pet(H, zebra)$
$\exists H : has\_drink(H, water)$

In order to complete the specification, implicit information must be made explicit and added to the specification:

The houses stand in a row:
$next(H1, H2) \rightarrow H1 = h1 \wedge H2 = h2 \vee H1 = h2 \wedge H2 = h3 \vee$
$H1 = h3 \wedge H2 = h4 \vee H1 = h4 \wedge H2 = h5$
$next(h1, h2) \wedge next(h2, h3) \wedge next(h3, h4) \wedge next(h4, h5)$

What means being *near*?
$near(H1, H2) \rightarrow next(H1, H2) \vee next(H2, H1)$
$next(H1, H2) \rightarrow near(H1, H2)$
$next(H1, H2) \rightarrow near(H2, H1)$

The predicates $has\_color/1$, $has\_nat/2$, $has\_prof/2$, $has\_drink/2$ and $has\_pet/2$ are typed:
$has\_color(H, C) \rightarrow house(H) \wedge color(C)$
$has\_nat(H, N) \rightarrow house(H) \wedge nat(N)$
$has\_prof(H, S) \rightarrow house(H) \wedge prof(S)$
$has\_drink(H, D) \rightarrow house(H) \wedge drink(D)$
$has\_pet(H, P) \rightarrow house(H) \wedge pet(P)$

We know what are the houses, the colours, the nationalities, the drinks, the professions and the pets:
$house(H) \rightarrow H = h1 \vee H = h2 \vee H = h3 \vee H = h4 \vee H = h5$
$house(h1) \wedge house(h2) \wedge house(h3)) \wedge house(h4)) \wedge house(h4)$
$color(C) \rightarrow C = red \vee C = yellow \vee$
$C = green \vee C = blue \vee C = white$
$color(red) \wedge color(yellow) \wedge color(green) \wedge color(blue) \wedge color(white)$
$pet(P) \rightarrow P = dog \vee P = horse \vee P = snails \vee P = fox \vee P = zebra$
$pet(dog) \wedge pet(horse) \wedge pet(snails) \wedge pet(fox) \wedge pet(zebra)$
$nat(N) \rightarrow N = norwegian \vee N = italian \vee N = japanese$
$\vee N = spaniard \vee N = englishman$

$nat(norwegian) \wedge nat(italian) \wedge nat(japanese)$
$\wedge nat(spaniard) \wedge nat(englishman)$
$prof(S) \rightarrow S = doctor \vee S = violinist \vee S = diplomat \vee$
$S = painter \vee S = sculptor$
$prof(doctor) \wedge prof(violinist) \wedge prof(diplomat) \wedge prof(painter)$
$\wedge prof(sculptor)$
$drink(D) \rightarrow D = wine \vee D = coffee \vee D = milk \vee D = tea \vee$
$D = water$
$drink(wine) \wedge drink(coffee) \wedge drink(milk) \wedge drink(tea) \wedge$
$drink(water)$

Finally, we must express that each house has one colour, nationality, profession, drink and pet:
$has\_color(H, C1), has\_color(H, C2) \rightarrow C1 = C2$
$has\_nat(H, N1), has\_nat(H, N2) \rightarrow N1 = N2$
$has\_prof(H, S1), has\_prof(H, S2) \rightarrow S1 = S2$
$has\_drink(H, D1), has\_drink(H, D2) \rightarrow D1 = D2$
$has\_pet(H, P1), has\_pet(H, P2) \rightarrow P1 = P2$
$house(H) \rightarrow \exists C, N, S, D, P : has\_color(H, C) \wedge has\_nat(H, N) \wedge$
$has\_prof(H, S) \wedge has\_drink(H, D) \wedge has\_pet(H, P)$

The problem is to find out the nationality, colour, drink, pet and profession of each house. Just as for the exam scheduling example in chapter 1, the solution of the problem is given by a model of the specification. Note that equality appears in the head and that two distinct constants represent always distinct objects. Therefore, the NMGE instance for FEQ as equality theory with completion applies. NMGE is able to solve the problem. However, the prototype shows an important efficiency problem, due to the clauses of the form:

$$nat(N) \rightarrow N = norwegian \vee N = italian \vee N = japanese \vee \ldots$$

NMGE generates for each derived fact $nat(sk)$ each of the five possibilities by naive backtracking. This results in a combinatorial explosion, which in cooperation with the other inefficiencies of the prototype results in the long execution time. This combinatorial explosion could be avoided by integrating constraint solving techniques in NMGE. The impact of such techniques is important, as was shown by [Van89], who presents a CLP solution for the puzzle.

Recently, important efforts were made in the context of the Japanese Fifth Generation Computer Systems project to implement an efficient model generator. This resulted in the implementation of MGTP, a efficient parallel model generator implemented on the Parallel Inference Machine [FHKF92]. A restriction of MGTP is that it does not provide special treatment for equality. Integrating our techniques for dealing with equality and techniques for constraints solving in NMTP

could result in an efficient procedure able to solve declaratively specified model generation problems.

In our opinion, the major distinction between our specification and a solution as in [Van89], even though the latter is less verbose than ours, is that our specification can be developed without taking in account the problem to be solved. If our problem would have been to know how many pets there are, or whether the Japanese occupies the middle house, or -in case the specification would have been inconsistent- which fact should be deleted to restore the consistency, for each of these problems, our domain specification would have been of use. The same can not be said of the solution in [Van89]. Here is -in our opinion- an essential distinction between a *specification* and a *program*. A logic specification can be constructed as a description of the domain knowledge, without looking at what problem is to be solved (except that relevant information must be present). In contrast, a program is built to solve a specific problem and encodes a problem specific representation of the problem domain.

It should be noted that the above specification can be simplified by a different representation. Note that the predicates $has\_nat/2$, $has\_color/2$, etc. represent one to one correspondences. Hence, these predicates could be represented by a functor, e.g. a functor $house\_of/1$ which maps a colour, nationality, pet drink or professions on houses. The formulation that the white house is near the blue one becomes $near(house\_of(white), house\_of(blue))$. In this representation, the axioms of FEQ do not apply any longer: e.g. $house\_of(white)$ is equal to one of the constants $h_1, \ldots, h_5$. Hence the instance of NMGE with EQ applies. Unfortunately, now lots (60) of disequality facts are needed: $h1 \neq h2$, $english man \neq italian$, etc.. What is needed here is a new instance of the framework in which distinction is made between *constructor* functors for which the axioms of FEQ hold, and *function* functors for which FEQ does not hold. In the above example, all constants representing houses, pets, etc.. are constructor constants, whereas $house\_of/1$ is a function functor. This is a subject for future research.

3.8 Discussion

The current duality framework is limited to definite (abductive) programs. How could the duality be extended to the normal case and to (abductive extensions of) SLDNF? A natural dual interpretation for SLDNF, which extends the duality of SLD, should interpret SLDNF derivations as model generations or at least as consistency proofs in the only-if part of the completion. When constructing a failure tree for some atom to prove its negation, Clark showed that SLDNF basically proves the negated atom using the only-if part of the completion. However, as its name says, a failure tree shows in the first place a failure to construct a refutation for the atom, hence it can be seen as a consistency proof of the negative literal wrt the if-part of the program. Or, SLDNF's reasoning using the if-part corresponds

to some form of model generation in the only-if part, while its reasoning in the only-if part corresponds to consistency proving in the if-part. This suggests that the dual theory of a normal abductive program should be the whole completion of the program, and that an SLDNF refutation could be dually interpreted as a form of model generation or at least consistency proof of $\neg Q + comp(P)$.

Unfortunately, this does not work. A trivial program shows this. Consider the program $P = \{r \text{:-} \neg r \quad q \text{:-} \neg p \quad p \text{:-} false\}$. The query $\leftarrow q$ has a successful SLDNF refutation. However, $comp(P) + \{q\}$ is not consistent (and the dual interpretation of SLDNF can definitely not be a model generation or consistency proving wrt $comp(P)$).

This unpleasant conclusion made us wonder whether the problem to extend the duality was not due to inherent problems of the completion semantics. It was this problem that has triggered our research on the semantics of abductive logic programs. The result of this work is presented in the next chapter. In the semantics that we define there, the duality will have a very nice extension, which goes as follows:

> Given is an abductive logic program P, a query Q and a set of abducible facts Δ.
>
> $P + \Delta \models \neg Q$ iff P and $\neg Q$ are consistent and have a model which extends Δ.
>
> Therefore, any sound abductive procedure which returns a solution Δ for a query Q, proves not only $P + \Delta \models \neg Q$ but also the consistency of $P + \Delta$. This implicit consistency proof can be seen as the dual interpretation of the abductive derivation and replaces model generation as dual interpretation of SLD+Abduction.

Related to our work, [Bry90] also indicates a relationship between abduction and model generation. The nature of this work differs from ours. The goal of [Bry90] is to develop an abductive procedure in the context of updating deductive databases. A meta-theory is proposed which takes a query and an abductive program P as input and, when executed by a model generator, generates abductive solutions. Our work takes the alternative approach of executing the model generator directly on *only-if*(P). This allows us to present a more explicit duality, not only on the level of the abductive solutions and the generated models, but also on the fine grained computational steps involved in the applied abduction and model generation procedures. The meta-approach of [Bry90] makes no reference to the only-if part of the abductive program. In addition, on a more technical level, no equality atoms appear in the head of the meta-program and therefore, no special treatment for FEQ is necessary.

In [CTT91], another approach is taken for abduction through deduction. An abductive procedure is presented which for a given normal abductive program P

and query $\leftarrow Q$, derives an *explanation formula* E equivalent with Q under the completion of P:

$$comp(P) \models (Q \Leftrightarrow E)$$

The explanation formula is built of abductive predicates and equality only. It characterises all abductive solutions in the sense that for any set Δ of abducible atoms, Δ is an abductive solution iff it satisfies E.

Although this approach departs also from the concept of completion, it is of a totally different nature. In the first place, our approach aims at contributing to the procedural semantics of abduction. This is not the case with the work in [CTT91]. Another difference is that this approach is restricted to queries with a finite computation tree. If the computation tree contains an infinite branch, then the explanation formula cannot be computed.

Finally, we want to draw attention to an unexpected application of the duality framework. An uncommon form of abduction is obtained if FEQ is replaced by general equality EQ and the equality predicate is abducible. This form of abduction is presented in [CEP92]. Take the program $P = \{r(a)\leftarrow\}$. For this program, the query $\leftarrow r(b)$ has a successful abductive derivation.

$\leftarrow \underline{r(b)}$ $\quad \Delta = \{\}$

$\square$ $\quad \Delta = \{b = a\}$

$\leftarrow r(b)$ succeeds under the abductive hypothesis $\{b = a\}$. The duality framework provides the technical support for efficiently implementing this form of abduction. The difference with normal abduction is that the completion procedure for FEQ -the dual of unification- must be replaced by a completion procedure for EQ, for example Knuth-Bendix completion.

3.9 Summary

We summarise the contributions of this chapter. The starting point was the question whether the well-known duality on the declarative level between abductive solutions for a definite abductive program and models of the only-if part of its completion, could be extended to the procedural semantics. Dualizing unification and application of mgus in the abductive procedure, we found techniques for dealing efficiently with equality in model generation. The dual techniques extend existing techniques found in Term Rewriting. The intuition of our approach is simple: during model generation in a theory with equality in the head of rules, sets of facts are generated which contain subsets of ground equality facts. By *contracting* the sets of generated facts wrt to their ground equality facts, a compact representation is obtained of the sets plus all their (possibly infinite) logic consequences under the

underlying equality theory. The contraction is performed by transforming the set of equality facts in a *complete term rewriting system* and *normalising* the generated non-equality facts wrt to this term rewriting system. The *completion* of the equality facts corresponds dually with the computation of the mgu. The *normalisation* corresponds dually with the application of the mgu on the resolvent. For theories with equality, we have shown that our NMGE procedure will often avoid the infinite loops which occur in an execution of the normal procedure. We have illustrated the potential that such a procedure might have for executing declarative specifications. Transferring the methods back to abduction, we will obtain techniques for efficient treatment of abduced equality atoms and for abduction under the standard equality theory (instead of Free Equality).

Chapter 4

A Semantics for Abductive Logic Programs

4.1 Introduction.

Traditionally, logic programming is situated somewhere halfway between declarative knowledge representation as in first order logic and procedural programming as in classical imperative languages. At this moment, a uniform view on the normal clause formalism as a declarative logic lacks. A large number of different semantics have been proposed [vEK76] [Cla78] [Fit85] [Kun89] [ABW88] [Prz88] [GL88] [VRS91] [Prz90] [KM90b] [Dun91] [KM90c] [CTT91] etc.. . Recently, a number of studies have been published which provide a first step towards more uniformity. These studies integrate some of the above semantics in a unifying semantic framework [Prz90] [BLMM92] [Bon92] [Dix92]. Such frameworks provide a formal way of comparing different semantics, showing relationships and dissimilarities. Here, we present a unifying framework for the semantics of normal abductive logic programs with FOL axioms. The framework covers and extends the currently most widely accepted families of semantics: completion semantics, stable and stationary semantics and well-founded semantics.

In general, programmers view their logic programs as *sets of definitions describing the truth of facts in terms of more primitive facts.* This view extrapolates to abductive logic programs: an abductive logic program can be considered as an incomplete definition set, containing a number of *undefined* predicates. This view motivated us to investigate how the truth values of facts in a model are constructed in diverse types of semantics. The framework is based on the concept of *justification.* A justification can be seen as a mathematical object justifying the truth value of facts in terms of truth values of other facts. Three different instances of the framework are obtained by defining three different notions of *justifications.*

A first notion of justification is found in completion semantics [Cla78]. In the completion semantics, a fact F is true iff it occurs in the head of a rule with a true body. This suggests the following definition for what we call a *direct justification* of F: a set of facts occurring in the body of a ground instance of a rule with head F. Not all direct justifications are successful: a direct justification which contains a false fact cannot be used to "justify" the truth of F. Since we allow 3-valued interpretations, we can associate one of the values *false* (**f**), *unknown* (**u**) or *true* (**t**) to a direct justification, depending on the minimal truth value of its elements. A *directly justified model* is defined as an interpretation in which the truth value of each fact is the value of its most successful direct justification.

In completion semantics, definitions are not *constructive*: a program $P_1 = \{p \text{:-} p\}$ has a model $\{p\}$, in which p is directly justified by itself. It is precisely this feature, which makes the model so counter intuitive for many people. Programs like P_1 and like the well-known transitive closure program, which contain *positive loops*, have motivated the development of other semantics, such as stable semantics [GL88] and well-founded semantics [VRS91]. Recently, [Fag90] exposed the notion of justification underlying the stable semantics: direct justifications can be concatenated to form trees of direct justifications. For a positive fact which is justified via an atomic rule we add *true* (denoted by ■) as descendant. For a positive fact which does not match with the head of any rule we add *false* (denoted by □) as descendant. By maximally extending justification trees, we obtain trees in which all leaves contain ■, □, undefined facts or negative defined facts $\neg F$ but no positive defined facts. In [Fag90], a justification is defined as the set of all non-root nodes of such trees. Instead, in our framework we use the trees as justifications and call them *partial justifications*. The partial justification of p in P_1 is $p \leftarrow p \leftarrow \ldots$. Now the cycle is apparent. It suffices to assign the value **f** to partial justifications with positive loops, to avoid the counter intuitive models. In general a partial justification which contains a false negative defined or undefined fact in a leaf or which contains an infinite branch of positive facts is assigned **f** as value. Otherwise, if it contains an unknown fact in a leaf, it is assigned **u** as value. Otherwise, its value is **t**. A justification with value **t** has finite depth and contains only true leaves.

There remains a class of problematic programs for which definitions are still not constructive, programs which are *looping over negation*. A standard example is $P_2 = \{p \text{:-} \neg q \;\; q \text{:-} \neg p\}$. This program has stable models $\{p\}$ and $\{q\}$. The justifying partial justification of p in the first model is $p \leftarrow \neg q$. The problem is that $\neg q$ itself depends on p. Or, stable semantics accept models with cyclic dependencies over negation. Well-founded semantics do not, and the unique well-founded model is $\{p^{\mathbf{u}}, q^{\mathbf{u}}\}$.

How can the notion of justification be refined to detect these cyclic dependencies? What is needed here is some structure which records dependencies of negative facts on other facts. Our solution is based on an extension of the notion of *direct*

justification for negative facts. A *direct negative justification* for a negative fact $\neg F$ is a set obtained by selecting one fact from the body of each ground instance of a rule whose head matches F, and adding the negation of this fact to the set. E.g. the unique direct negative justification for $\neg q$ in P_2 is $\{p\}$. This set records a one level dependency of $\neg q$ on p. We define a *justification* as a maximal tree obtained by concatenating direct positive and direct negative justifications. Such trees have only $\square$, $\blacksquare$ and undefined facts in the leaves. In the program P_2, the loop over negation for p becomes apparent in the justification $p \leftarrow \neg q \leftarrow p \leftarrow \ldots$. This notion of justification turns out to be very similar to the notion of WFM-tree, which was introduced in [PAA91a] in the context of a derivation procedure for propositional logic programs under the well-founded semantics.

For a positive fact F to be true, intuitively one requires that it can be associated a loop-free partial justification with true leaves and, moreover, that this partial justification should be extendible to a justification which contains no loop over negation. Also, this justification should comprise for each contained positive fact its associated loop-free partial justification. Justifications satisfying these conditions are assigned the value **t**. Justifications that contain correct partial justifications but loops over negation, are assigned the value **u**. Note that a justification with value **t** may contain *negative loops*: if we add the rule q :- $\neg p$ to the program P_1, then q has justification $q \leftarrow \neg p \leftarrow \neg p \leftarrow \ldots$, which contains a negative loop. The resulting justified model is $\{q^{\mathbf{t}}\}$[1]. In general, we obtain the following definition: the value of a justification with a false leaf or with a positive loop is **f**. Otherwise, if the justification contains an unknown leaf or a loop over negation, its value is **u**. Otherwise we define its value as **t**. Justifications with value **t** have only true leaves and all infinite branches are negative loops.

The framework is based on 3-valued general interpretations and is defined for abductive logic programs. Not only does it cover the existing semantics but it also incorporates extensions of them, e.g. a new 3-valued completion semantics for abductive logic programs; e.g. justification semantics as an extension of well-founded semantics for abductive programs with general interpretations. As a result, the framework still augments the abundance on different semantics. However, by making explicit how true facts are constructed in different semantics, the framework also shows that not all semantics are equal implementations of the view of programs as sets of *constructive definitions*. Well-founded semantics and its extension, justification semantics, are the only semantics in which a true positive fact never depends on itself. Or, well-founded and justification semantics provide the most constructive formalisation of logic programs as sets of definitions.

[1] That negative loops are allowed in true justifications, makes apparent this remarkable asymmetry between positive facts and negative facts which is present in the closed world assumption and in virtually all semantics for logic programs, starting from the Least Herbrand Model semantics: a negative fact needs no proof: it suffices that it is consistent with the theory to be true. Therefore a true negative fact may depend on itself.

A part of our work (sections 4.7, 4.8, 4.9) is devoted to exploring this view of logic programs as constructive definitions (in the sequel the *constructive definition view*). From this point of view, a program with a loop over negation, like P_2, makes no sense. A rational solution would be to define the program as contradictory. The solution in well-founded semantics and justification semantics of assigning such facts truth value **u** seems better. Often inconsistencies of this type are located in a small part of the program, and the rest of the program will have a sensible meaning. Therefore, the use of **u** to allow *local inconsistencies* is a better, more permissive solution because it enables graceful degradation of information retrieval in the presence of inconsistent data[2]. Therefore, we propose to interpret **u** as *locally inconsistent* instead of the weaker *unknown*. This does not mean that no uncertainty can be represented in the formalism. As a matter of fact, an important theorem of the framework is that any first order theory can be transformed to a logically equivalent abductive logic program (wrt justification semantics). The expressivity of FOL for representing uncertainty is widely accepted. One successful experiment of the representation of incomplete knowledge in abductive logic programming, in which we have been mostly interested, is in temporal reasoning: abductive event calculus has been successfully applied for AI-planning and temporal reasoning under uncertainty [Esh88], [Sha89], [DMB92] (see also chapters 6 and 7).

The chapter is structured as follows. In sections 4.2 and 4.3, we define three notions of *justifications* and the associated notions of model and investigate the properties of the resulting semantics. In section 4.4, the relationship with completion semantics is elaborated. In section 4.5, the relationship with stable, stationary and well-founded model semantics is shown. In section 4.6, we present a duality theorem: a theorem which extends the work in the previous chapter for normal abductive programs. In section 4.7, the constructive definition view is compared with the popular view of negation as negation by default or negation as abduction. The two following sections 4.8 and 4.9 explore different aspects of the constructive definition view, such as the nature of negation and the representation of incomplete knowledge. The chapter ends with a conclusion (section 4.11). A short paper on these results was published as [DD93a].

4.2 Justification Semantics for logic programs

We define the semantics of logic theories $\mathcal{T}$ consisting of a normal logic program $\mathcal{T}_d$ and a theory $\mathcal{T}_c$ of FOL axioms. The semantics of $\mathcal{T}_c$ is as defined in chapter 2. The semantics of the logic programs is based on the concept of justification: a

[2] In the presence of inconsistent data (e.g. $p \wedge \neg p$), a first order theory entails any formula. This abrupt collapse has been considered by many as a serious problem of classical logic, e.g. see [THT87]. The introduction of local inconsistency in logic provides an adequate answer to this critique.

defined fact is true if and only if it has a justification.

Below we extend each language $\mathcal{L}$ with new propositional predicates $\blacksquare$ and $\square$. We extend each interpretation I such that $\mathcal{H}_I(\blacksquare) = \mathbf{t}$ and $\mathcal{H}_I(\square) = \mathbf{f}$. For notational simplicity, we define a negation operator $\sim$ on the set of positive and negative facts: if F is a positive fact, then $\sim F$ is defined as $\neg F$. If F is a negative fact $\neg F'$, then $\sim F$ is defined as F'. Observe that $\neg\neg F$ is not identical to F, whereas $\sim\sim F$ and F are identical. $\sim$ can be extended to sets and sequences of facts. We define $\sim\blacksquare = \square$ and vice versa. The semantics defined in the following section are based on 3-valued general interpretations. In the sequel of this chapter we will denote normal atomic clauses $p(\overline{t})$:- by $p(\overline{t})$:-$\blacksquare$.

Recall from section 2.1 that for a ground fact $F = p(t_1, \ldots, t_n)$, $\tilde{I}$(F) denotes the simple fact $p(\tilde{I}(t_1), \ldots, \tilde{I}(t_n))$.

Definition 4.2.1 (Direct Positive justification) *Given is a language $\mathcal{L}$, an interpretation I, a logic program $\mathcal{T}_d$ and a simple positive fact F of a defined predicate p/n.*

A set J is called a direct positive justification ($\mathcal{DPJ}$) for F if there exists a ground instance F' :- $F_1, .., F_k$ of a rule of the definition of p/n with $\tilde{I}(F') = F$ and $J = \{\tilde{I}(F_1), .., \tilde{I}(F_k)\}$.

If no such ground instance exists for F, then we call $\{\square\}$ a direct positive justification of F.

The intuition for allowing $\{\square\}$ as a justification, is that F can only be true if $\square$ is true, i.e. in case of inconsistency. Due to this trick, each simple positive defined fact has a direct justification, and a direct justification is never empty (due to our notation for atomic rules A :-$\blacksquare$). A direct positive justification is always finite.

Example Consider a transitive closure program:

$$
\begin{aligned}
P_{trans} = \{\ & tr(X,Y) \text{:-} p(X,Y) \\
& tr(X,Z) \text{:-} p(X,Y), tr(Y,Z) \\
& p(a,a) \text{:-} \blacksquare \\
& p(b,c) \text{:-} \blacksquare
\end{aligned}
$$

In any Herbrand interpretation, the fact $tr(a,b)$ has the following direct justifications: $\{p(a,b)\}$, $\{p(a,a), tr(a,b)\}$, $\{p(a,b), tr(b,b)\}$, $\{p(a,c), tr(c,b)\}$.

The counterpart of a $\mathcal{DPJ}$ for a negative fact is the direct negative justification for negative facts.

Definition 4.2.2 (Direct Negative Justification) *A set J is called a direct negative justification (a $\mathcal{DNJ}$) for a negative simple fact $\neg F$ of a defined predicate iff it is obtained by selecting from each $\mathcal{DPJ}$ J' of F a fact G and adding $\sim G$ to J. Formally:*

- *for each $\mathcal{DPJ}$ J' of F: $\sim J' \cap J \neq \phi$*
- *for each $F' \in J$, there exists a $\mathcal{DPJ}$ J' of F: $\sim F' \in J'$*

A direct justification ($\mathcal{DJ}$) is a $\mathcal{DPJ}$ for a positive fact and a $\mathcal{DNJ}$ for a negative fact.

Example One of the 8 $\mathcal{DNJ}$'s for $\neg tr(a,b)$ in P_{trans} is:

$$\{\neg p(a,b), \neg tr(a,b), \neg p(a,c)\}$$

Analogously as for positive facts, each simple negative fact $\neg F$ has a direct negative justification and a direct negative justification is never empty. A direct negative justification can be infinite. For example, consider the language with functors $0, f/1$ and predicates $p/0, q/1$. The program consists of the rules

$$\{p \text{ :- } q(X) \ ; \quad q(X) \text{ :- } \Box\}$$

The least Herbrand model of this program is ϕ. The fact $\neg p$ has an infinite direct negative justification $\{\neg q(x) | x \in HU\}$.

Before we define the semantics associated with direct justifications, we introduce the two other notions of justifications, called *partial justifications* and *justifications*. Both types are special cases of trees obtained by concatenating direct justifications. In general, we call such a tree an *open justification.*

Definition 4.2.3 (Justifications) *An open justification J for a simple fact F is a (possibly infinite) tree of simple facts with F in the root. Each non-leaf node contains a defined fact such that the set of descendants of the node form a direct (positive or negative) justification for F.*

A partial positive justification ($\mathcal{PPJ}$) for a positive defined simple fact F is an open justification for F such that all non-leaves contain positive defined facts and no leaf contains a positive defined fact.

A partial negative justification ($\mathcal{PNJ}$) for a negative defined simple fact F is an open justification for F such that all non-leaves contain negative defined facts and no leaf contains a negative defined fact.

A justification ($\mathcal{J}$) J for F is an open justification for F such that no leaf contains a defined fact.

Example In any Herbrand interpretation of P_{trans}, $tr(a,b)$ has a unique partial justification without $\Box$, but it contains an infinite branch of $tr(a,b)$ facts. This is shown in figure 4.1. The negative fact $\neg tr(a,b)$ has also partial justification without $\Box$. It contains an infinite branch of negative facts $\neg tr(a,b)$. This partial justification is shown in figure 4.2.

Both the partial justifications for $tr(a,b)$ and $\neg tr(a,b)$ are examples of justifications since they do not contain defined facts of the opposite sign in the

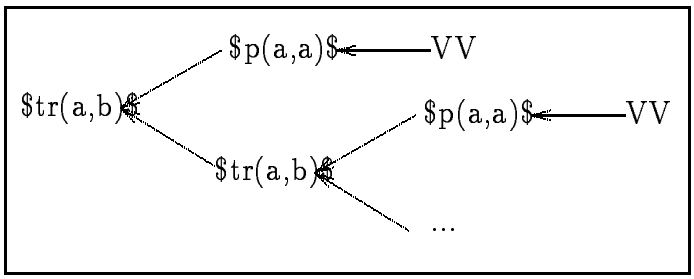

Figure 4.1: Partial Positive Justification of $tr(a, b)$

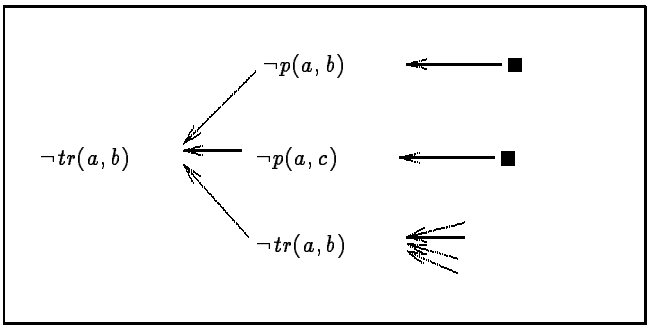

Figure 4.2: Partial Negative Justification of $\neg tr(a, b)$

leaves. Now we add the clause q :- $\neg tr(a, b)$ to P_{trans}. q has a partial justification $p \leftarrow \neg tr(a, b)$. It has a justification obtained by concatenating the justification of $\neg tr(a, b)$ in figure 4.2 to the partial justification of q. This justification is shown in figure 4.3.

Example Consider the following program in which r is undefined:

$$P = \{ \begin{array}{l} p \text{ :- } r, \neg q \\ q \text{ :- } r, \neg p \ \} \end{array}$$

Figure 4.4 gives a justification for p in an interpretation in which r is true. Note that it contains an infinite branch with an infinite number of positive and negative facts.

In appendix B, we show that the concept of open justification is well-defined

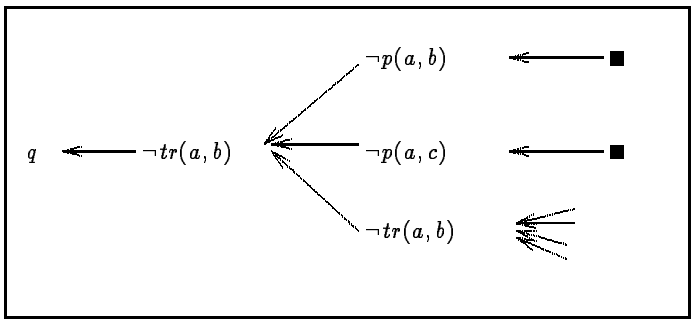

Figure 4.3: Justification of p

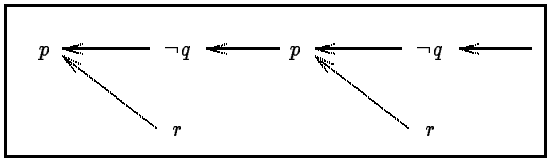

Figure 4.4: Justification of p

and that the set of justifications is equipped with a partial order which is defined as follows:

$J_1 < J_2$ iff J_2 is obtained by extending J_1 at the leaves.

Moreover, $<$ satisfies the important property, that each monotonically increasing sequence of open justifications has a LUB or a fixpoint. On the set of open justifications, a concatenation operation is defined. Formally, this operation can be defined as follows. Let J be an open justification, S a subset of the leaves of J, h a mapping with domain S which maps a node N in S to an open justification of the label of N. We define the concatenation of J and h as the extension of J obtained by attaching $h(N)$ to J in N for each N in S.

In the sequel, we distinguish between a *path* and a *branch* in an open justification J. A path is a sequence of facts $(F_0, F_1, \ldots)$ with F_0 in the root of J, and each F_i a descendant of F_{i-1} in J. A branch is a maximal path. A *positive loop* is a branch with an infinite number of positive facts and a finite number of negative facts. An example is the sequence of $tr(a,b)$ facts in figure 4.1. A *negative loop* is a branch with an infinite number of negative facts and a finite number of positive facts. Examples are the branches with an infinite number of $\neg tr(a,b)$ facts

in figures 4.2 and 4.3. A *loop over negation* is a branch with an infinite number of positive and negative facts. An example is the branch with an infinite number of p and $\neg q$ facts in figure 4.4. J is looping if it contains a positive loop or a loop over negation. Otherwise it is called loop-free (observe that loop-free justifications may contain negative loops).

In the context of building a $\mathcal{PPJ}$, we call the leaves of an open justification with positive defined facts *open leaves* and the branches leading to them *open branches*. In the context of building a $\mathcal{PNJ}$, the leaves with negative defined facts and the branches leading to such leaves are called *open*. In the context of building a $\mathcal{J}$, leaves with positive and negative defined facts and branches leading to them are called *open*.

Next we define the value of a justification as a measure for its success.

Definition 4.2.4 (value of a justification) *Let I be an interpretation.*

Let B be a branch. If B is finite and has F as leaf then the value of B under I is $\mathcal{H}_I(F)$. With respect to infinite branches, we define the value of a positive loop as **f**, *the value of a loop over negation as* **u** *and the value of a negative loop as* **t**. *We denote the value of B under I by $val_I(B)$.*

Let J be an open justification. The value of J under I is $min\{val_I(B)|B$ is a branch of $J\}$. We denote J's value by $val_I(J)$. J is false, weak, strong under I if $val_I(J)$ is **f**, **u**, **t** *respectively.*

Observe that for any branch B, $val_I(\sim B) = val_I(B)^{-1}$.

The essential idea in our semantics is that an interpretation is a model of a logic program P iff for each defined positive simple fact F, its truth value is equal to the value of its most successful (direct justification)(partial justification)(justification). We call these values *the supported values* of F wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$ and denote them by $SV_{DJ}(P,I,F)$, $SV_{PJ}(P,I,F)$, $SV_J(P,I,F)$ respectively. Here $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$ stand for (direct justification)(partial justification) (justification) semantics. Formally $(SV_{DJ}(P,I,F))$ $(SV_{PJ}(P,I,F))$ $(SV_J(P,I,F))$ denote $max\{val_I(J)|J$ is a $(\mathcal{DJ})$ $(\mathcal{PJ})$ $(\mathcal{J})$ of F in $P\}$. In general, when P is clear from the context, we drop it as an argument, writing $SV_{DJ}(I,F)$, $SV_{PJ}(I,F)$, $SV_J(I,F)$.

Definition 4.2.5 *A (directly justified) (partially justified) (justified) model of a logic program $\mathcal{T}_d$ is an interpretation I of $\mathcal{L}$ such that for every simple positive defined fact F:*

$$\mathcal{H}_I(F) = (SV_{DJ}(I,F))(SV_{PJ}(I,F))(SV_J(I,F))$$

Moreover the interpretation of undefined predicates is two-valued.

A (directly justified)(partially justified)(justified) model of a theory $\mathcal{T}$ consisting of a logic program $\mathcal{T}_d$ and FOL axioms $\mathcal{T}_c$ is a (directly justified)(partially justified)(justified) model of $\mathcal{T}_d$ and a model of $\mathcal{T}_c$.

Definition 4.2.6 (Entailment) *Given is some theory $\mathcal{T}$ and closed formula F both based on $\mathcal{L}$.*

We define $\mathcal{T} \models_{\mathcal{DJS}} F$ iff for any directly justified model M of $\mathcal{T}$, $\mathcal{H}_M(F) = \mathbf{t}$. We define $\mathcal{T} \models_{\mathcal{PJS}} F$ and $\mathcal{T} \models_{\mathcal{JS}} F$ in the analogous way.

Example Let $\mathcal{L}$ be the language with functors 0/0 and f/1, and predicate $p/1$.

$$P_2 = \{\ p(X) \text{ :- } p(f(X))\ \}$$

A fact $p(f^n(0))$ in a Herbrand interpretation has the direct justification:

$$p(f^n(0)) \leftarrow p(f^{n+1}(0))$$

A directly justified Herbrand model is:

$$\{p(f^n(0))^{\mathbf{t}} | n \in \mathbb{N}\}$$

This interpretation is not a partially justified or justified model. E.g. $p(0)$ has (partial) justification $p(0) \leftarrow p(f(0)) \leftarrow \ldots$, which contains a positive loop. Hence, the only (partially) justified Herbrand model is the empty Herbrand interpretation. This example shows that a positive loop is not always caused by cyclic dependencies.

Example The transitive closure program P_{trans} has a unique partially justified Herbrand model, in which $tr(a, b)$ is false. However, there exists a non-Herbrand model in which $tr(a, b)$ is true:

$$(D = \{x\}, \{a \rightarrow x, b \rightarrow x, c \rightarrow x\}, \{p(x, x)^{\mathbf{t}}, tr(x, x)^{\mathbf{t}}\})$$

$p(x, x)$ has the strong partial justification $p(x, x) \leftarrow \blacksquare$, using for example the atomic rule $p(a, a)$:- $\blacksquare$ of P. $tr(x, x)$ has the strong partial justification $tr(x, x) \leftarrow p(x, x) \leftarrow \blacksquare$, using the rule $tr(X, Y)$:- $p(X, Y)$. In order to avoid this undesired model, FEQ must be added.

Example Let $\mathcal{L}$ again be the language with functors 0/0 and f/1, and predicate $p/1$.

$$P_3 = \{\ p(X) \text{ :- } \neg p(f(X))\ \}$$

A fact $p(f^n(0))$ has direct justification $p(f^n(0)) \leftarrow \neg p(f^{n+1}(0))$. Two directly justified and partially justified Herbrand models are:

$$\{p(f^{2\times n}(0))^{\mathbf{t}} | n \in \mathbb{N}\}$$

$$\{p(f^{2\times n+1}(0))^{\mathbf{t}} | n \in \mathbb{N}\}$$

These interpretations are not justified models. E.g. $p(0)$ in the first interpretation has justification $p(0) \leftarrow \neg p(f(0)) \leftarrow p(f^2(0)) \leftarrow \ldots$ which contains a loop over negation. Hence, the only justified Herbrand model is:

$$\{p(f^n(0))^{\mathbf{u}} | n \in \mathbb{N}\}$$

This example shows that a loop over negation is not always caused by cyclic dependencies.

It is absolutely indispensable that undefined predicates have two-valued interpretations. Consider a trivial theory $\mathcal{T}_0$ with empty definition set and with FOL axioms $\{p\}$. The least one expects is that this theory entails p. If we would allow 3-valued interpretations for p, then p is not entailed because there is a weak model where p is not true. We could weaken the notion of entailment, and say that $\mathcal{T} \models F$ iff F is at least $\mathbf{u}$ in all models of P. Then $\mathcal{T}_0$ entails p but other undesirable effects pop up. Extend $\mathcal{T}_0$ with $p \rightarrow q$. In any case we expect that the new theory entails q. However, $\{p^{\mathbf{u}}, q^{\mathbf{f}}\}$ is a weak model in which q is false.

In the sequel, we extend the notion of supported value to undefined facts and negative facts. The supported value under I of an undefined fact F is defined as $\mathcal{H}_I(F)$. The supported value under I of a negative simple fact F can be defined analogously as for positive simple facts: it is equal to the value of the best (direct) (partial) justification of F.

4.3 Consistency and relationships.

The first theorem is of fundamental importance in our theory. It asserts that a fact and its negation have inverse supported values in the three types of semantics.

Theorem 4.3.1 (consistency) *Let I be an interpretation, F be a simple fact.*
(a) $SV_{DJ}(I, F) = SV_{DJ}(I, {\sim}F)^{-1}$
(b) $SV_{PJ}(I, F) = SV_{PJ}(I, {\sim}F)^{-1}$
(c) $SV_{J}(I, F) = SV_{J}(I, {\sim}F)^{-1}$

A direct consequence of this theorem is that in a model of each type, the truth value of negative facts is also equal to their supported value. This restores the asymmetry between positive and negative facts in the definition of model which requires only that positive facts have truth value equal to their supported value. In the future, when we refer to *a fact* F, F may be both positive or negative, unless explicitly indicated.

Despite the conceptual simplicity of the theorem, its proof turns out to be tedious. It uses the following lemma.

Lemma 4.3.1 *Let F be a simple fact.*

(a) For any direct justification J_1 for F and J_2 for $\sim F$: $J_1 \cap \sim J_2 \neq \phi$.

(b) For any partial justification J_1 for F and J_2 for $\sim F$, there exist branches B_1, B_2 in J_1, J_2 resp. such that $B_2 = \sim B_1$.

(c) For any justification J_1 for F and J_2 for $\sim F$, there exist branches B_1, B_2 in J_1, J_2 resp. such that $B_2 = \sim B_1$.

Proof (a) is trivial by definition of $\mathcal{DNJ}$.

(b) and (c) can be proven in a similar way by applying (a) repeatedly. We prove only (b). Assume that F is a positive fact. We build two sequences of paths (B_i) and (B_i') in J_1 and J_2. For each i, B_i and B_i' are paths of length i in J_1 and J_2 resp. If $i > 0$, then B_i, B_i' are extensions of B_{i-1}, B_{i-1}' respectively and $B_i = \sim B_i'$.

We define B_1 and B_1' as paths of length 1, containing $F, \sim F$ respectively. Assume that we obtained B_i and B_i' for some $i \geq 0$. The terminal nodes of B_i and B_i' are labelled with F' and $\sim F'$. If F is an undefined fact, so is $\sim F'$; if F' is a negative defined fact, then $\sim F'$ is a positive defined fact. In both cases, F and $\sim F'$ are leaves of J_1 and J_2 resp. and the lemma follows from the induction hypothesis.

Assume that the terminal facts of B_i and B_i' are not leaves of J_1 and J_2. Now we can apply (a) of the lemma. Consider the direct justifications Jd_1 of F' in J_1, and Jd_2 of $\sim F'$ in J_2. There exists a fact F'' in Jd_1, such that $\sim F''$ is a fact in Jd_2. Define B_{i+1} as the path leading to F'', B_{i+1}' as the path leading to $\sim F''$. Clearly $B_{i+1}' = \sim B_{i+1}$.

□

Proof of theorem 4.3.1

(a) Let F be a positive simple fact. Consider a maximal $\mathcal{DPJ}$ J for F: $val_I(J) = SV_{DJ}(I, F)$. Every $\mathcal{DNJ}$ J' for $\sim F$ contains the negation of a fact from J. Therefore,

$$\begin{aligned} val_I(J') &\leq max\{\mathcal{H}_I(\sim G) | G \in J\} \\ &= (min\{\mathcal{H}_I(G) | G \in J\})^{-1} \\ &= (val_I(J))^{-1} \\ &= SV_{DJ}(I, F)^{-1} \end{aligned}$$

So $SV_{DJ}(I, \neg F) = max(\{val_I(J') | J' \text{ is a } \mathcal{DNJ} \text{ of } \neg F\}) \leq SV_{DJ}(I, F)^{-1}$.

To find the reverse inequality, we construct a $\mathcal{DNJ}$ J for $\sim F$ in the following way: for every $\mathcal{DPJ}$ J' of F, select a fact from J' with minimal truth value and add its negation to J. The value of J can be computed as follows:

$$val_I(J) = min\{\mathcal{H}_I(\sim G) | \exists J' : J' \text{ is a } \mathcal{DPJ} \text{ of F: } val_I(J') = \mathcal{H}_I(G)\}$$

$$
\begin{aligned}
&= (max\{\mathcal{H}_I(G) | \exists J' : J' \text{ is a } \mathcal{DPJ} \text{ of F: } val_I(J') = \mathcal{H}_I(G)\})^{-1} \\
&= (max\{val_I(J') | J' \text{ is a } \mathcal{DPJ} \text{ of } F\})^{-1} \\
&= SV_{DJ}(I, F)^{-1}
\end{aligned}
$$

Therefore: $SV_{DJ}(I, \neg F) \geq val_I(J) = SV_{DJ}(I, F)^{-1}$.

(b) The proof of (b) consists of three steps. In the first step we show for any simple fact F, $SV_{PJ}(I, F) \leq SV_{PJ}(I, \sim F)^{-1}$. This implies that if $SV_{PJ}(I, F) = \mathbf{t}$ then $SV_{PJ}(I, \sim F) = \mathbf{f}$. In the second step, we show that for any positive simple fact F, if $SV_{PJ}(I, F) \leq \mathbf{u}$ then $SV_{PJ}(I, \neg F) \geq \mathbf{u}$. These two items together imply directly that for a simple positive fact F, if $SV_{PJ}(I, F) = \mathbf{u}$, then $SV_{PJ}(I, \neg F) = \mathbf{u}$: indeed, by the first $SV_{PJ}(I, \neg F) \geq \mathbf{u}$; $SV_{PJ}(I, \neg F)$ cannot be equal to $\mathbf{t}$; otherwise by the first item, $SV_{PJ}(I, F)$ was necessarily $\mathbf{f}$. In the third step, we show that for any positive simple fact F, if $SV_{PJ}(I, F) = \mathbf{f}$ then $SV_{PJ}(I, \sim F) = \mathbf{t}$.

The three items prove (b) for positive facts F. The proof of (b) for negative facts can be derived directly from the case for positive facts.

Consider the first step. Let F be a defined fact and consider a maximal $\mathcal{PJ}$ J of $\sim F$. Every $\mathcal{PJ}$ J' of F comprises the negation of a branch B of J (lemma 4.3.1). As in (a):

$$
\begin{aligned}
val_I(J') &\leq val_I(\sim B) \\
&\leq max\{val_I(\sim B') | B' \text{ is a branch of } J\} \\
&= (min\{val_I(B') | B' \text{ is a branch of } J\})^{-1} \\
&= (val_I(J))^{-1} \\
&= SV_{PJ}(I, \sim F)^{-1}
\end{aligned}
$$

Hence, $SV_{PJ}(I, F) = max(\{val_I(J') | J' \text{ is a } \mathcal{PJ} \text{ of } F\}) \leq SV_{PJ}(I, \sim F)^{-1}$

The second step is more difficult to prove. Let F be a positive fact such that $SV_{PJ}(I, F) \leq \mathbf{u}$. We construct a monotonically increasing sequence (J_i) of open justifications which converges to a $\mathcal{PNJ}$ for $\neg F$ which is at least weak. Each J_i satisfies the following conditions:

(i) J_i is an open justification for $\neg F$ with depth $\leq i$.

(ii) All non-leaves in J_i contain negative defined facts and all leaves containing negative defined facts occur at depth i.

(iii) All undefined leaves and positive defined leaves in J_i are at least unknown.

(iv) For each defined negative fact $\neg F'$ in a leaf, $SV_{PJ}(I, F') \leq \mathbf{u}$.

Define J_0 as the tree with as root and only node $\neg F$. J_0 satisfies the conditions in a trivial way. We construct J_{i+1} by simultaneously extending all

open leaves of J_i containing a negative fact $\neg F'$ with a $\mathcal{DNJ}$ J_{dn}. We must select from each $\mathcal{DPJ}$ J_{dp} of F' one fact G and add $\sim G$ to J_{dn}. G is selected as follows: if J_{dp} contains an undefined or negative defined fact which is false or unknown, then we select this fact. Otherwise, since F' has no strong $\mathcal{PPJ}$, there must exist a positive defined fact in J_{dp} which has no strong $\mathcal{PPJ}$. We select this fact as G.

One easily verifies that J_{i+1} satisfies the four conditions. This sequence is monotonically increasing, hence it has a fixpoint J, which is obviously a $\mathcal{PNJ}$ for $\neg F$, and since all leaves are at least unknown, J is at least weak.

The third step is the case that a positive F has only false $\mathcal{PPJ}$'s. In a very similar way as the second step, we construct a monotonically increasing sequence (J_i) of open justifications for $\neg F$, which has a $\mathcal{PNJ}$ as limit. Each J_i satisfies the following conditions:

(i) J_i is an open justification for $\neg F$ with depth $\leq i$.

(ii) All non-leaves in J_i contain negative defined facts and all leaves containing negative defined facts occur at depth i.

(iii) All undefined leaves and positive defined leaves in J_i are true.

(iv) For each defined negative fact $\neg F'$ in a leaf, $SV_{PJ}(I, F') = \mathbf{f}$.

Define J_0 as the tree with as root and only node $\neg F$. J_0 satisfies the conditions in a trivial way. We construct J_{i+1} by simultaneously extending all open leaves of J_i containing a negative fact $\neg F'$ with a $\mathcal{DNJ}$ J_{dn}. We must select from each $\mathcal{DPJ}$ J_{dp} of F' one fact G and add $\sim G$ to J_{dn}. G is selected as follows: if J_{dp} contains an undefined or negative defined fact which is false, then we select this fact. Otherwise, since F' has only false $\mathcal{PPJ}$'s, there must exist a positive defined fact in J_{dp} which has only false $\mathcal{PPJ}$'s. We select this fact as G.

As for the second case, J_{i+1} satisfies the four conditions. The fixpoint J is a strong $\mathcal{PNJ}$ of $\neg F$. This finishes the proof of (b).

(c) The proof of (c) is very similar to (b). As in (b), we find from lemma 4.3.1 that for each simple fact F, $SV_J(I, F) \leq SV_J(I, \sim F)^{-1}$. Below we will proof that for any positive or negative simple fact F, if $SV_J(I, F) \leq \mathbf{u}$ then $SV_J(I, \sim F) \geq \mathbf{u}$. This suffices for the theorem. Indeed, from the first item it follows that if $SV_J(I, F) = \mathbf{t}$ then $SV_J(I, \sim F) = \mathbf{f}$; if $SV_J(I, F) = \mathbf{u}$ then from the first item $SV_J(I, \sim F) \geq \mathbf{u}$ but cannot be $\mathbf{t}$ since otherwise by the first item $SV_J(I, F) = SV_J(I, \sim\sim F) = \mathbf{f}$. If $SV_J(I, F) = \mathbf{f}$, then by the second item, $SV_J(I, \sim F) \geq \mathbf{u}$, but $SV_J(I, \sim F) = \mathbf{u}$ is impossible, since we just showed that then $SV_J(I, F) = SV_J(I, \sim\sim F) = \mathbf{u}$. Hence, $SV_J(I, \sim F) = \mathbf{t}$.

It remains to prove that if $SV_J(I,F) \leq \mathbf{u}$, then $\sim F$ has at least a weak $\mathcal{J}$. In a first step, we show that if $SV_J(I,F) \leq \mathbf{u}$, then there exists a loop-free partial justification J_p for $\sim F$ such that all undefined leaves of J_p are at least unknown and for all defined leaves F', $SV_J(I,\sim F')) \leq \mathbf{u}$.

Using this result, the proof of the theorem can then be terminated easily as follows. Observe that for each of the leaves F', $\sim F'$ satisfies the condition as F in the first step, namely $SV_J(I,\sim F')) \leq \mathbf{u}$. Hence we can re-apply the first step on the leaves of the $\mathcal{PPJ}$. We can continue to do so, and this process yields a sequence (J_i) of open justifications which converges to a justification J of $\sim F$ with all undefined leaves at least unknown and no positive loops. Hence, we find $SV_J(I,\sim F) \geq \mathbf{u}$.

The proof of the first step is split up in two cases, depending on the sign of F. We first consider the case that F is a positive fact. Assume $SV_J(I,F) \leq \mathbf{u}$. We construct a monotonically increasing sequence (J_i) which converges to a $\mathcal{PNJ}$ J_p of $\neg F$. All undefined leaves of J_p are at least unknown. For each positive defined leaf F', $SV_J(I,\neg F')) \leq \mathbf{u}$. The construction is very similar as in (b). Each J_i satisfies the following conditions:

(i) J_i is an open justification for $\neg F$ with depth $\leq i$.

(ii) All non-leaves in J_i contain negative defined facts and all leaves containing negative defined facts occur at depth i.

(iii) All undefined leaves in J_i are at least unknown.

(iv) For each positive or negative defined leaf F', $SV_J(I,\sim F') \leq \mathbf{u}$.

Define J_0 as usual. We construct J_{i+1} by simultaneously extending all open leaves of J_i containing a negative fact $\neg F'$ with a $\mathcal{DNJ}$ J_{dn}. J_{dn} is constructed by selecting from each $\mathcal{DPJ}$ J_{dp} of F' one fact G and adding $\sim G$ to J_{dn}. If J_{dp} contains an undefined fact which is false or unknown, then we select this fact as G. Otherwise, since $SV_J(I,F') \leq \mathbf{u}$, there must exist a positive or negative defined fact in J_{dp} whose supported value $\leq \mathbf{u}$. We select this fact as G. Clearly, J_{i+1} satisfies the conditions.

The fixpoint of (J_i) is a $\mathcal{PNJ}$ for $\neg F$, such that all undefined leaves are at least unknown and all defined leaves are positive and their negations have supported value $\leq \mathbf{u}$.

Now take the second case in which for a negative fact $\neg F$, it is given that $SV_J(I,\neg F) \leq \mathbf{u}$. We must show that F has a loop-free $\mathcal{PPJ}$, with all undefined leaves at least unknown and for all negative defined leaves $\neg F'$, $SV_J(I,F') \leq \mathbf{u}$. We proceed here by assuming the converse, that F has no such a $\mathcal{PPJ}$ and then constructing a strong $\mathcal{J}$ for $\neg F$.

By assumption, each $\mathcal{PPJ}$ of F has a false undefined leaf or contains a positive loop or contains a negative defined leaf $\neg F'$ such that $SV_J(I,F') =$

$\mathbf{t}$. We construct a $\mathcal{PNJ}$ J_p for $\neg F$ such that all undefined leaves are true and all positive defined leaves have a strong $\mathcal{J}$. Using J_p, it is straightforward to construct a strong $\mathcal{J}$ for $\neg F$. This contradicts $SV_J(I, \neg F) \leq \mathbf{u}$.

J_p is constructed as the fixpoint of a monotonically increasing sequence (J_i). Each J_i satisfies the following conditions:

(i) J_i is an open justification for $\neg F$ with depth $\leq i$.

(ii) All non-leaves in J_i contain negative defined facts and all leaves containing negative defined facts occur at depth i.

(iii) For all undefined facts or positive defined facts F' in a leaf, it holds that $SV_J(I, F') = \mathbf{t}$

(iv) For each defined negative fact $\neg F'$ in a leaf, F' satisfies the same condition as F, namely each $\mathcal{PPJ}$ of F' has a false undefined leaf or contains a positive loop or contains a negative defined leaf $\neg F'$ such that $SV_J(I, F') = \mathbf{t}$.

Define J_0 as usual. We construct J_{i+1} by simultaneously extending all open leaves of J_i containing a negative fact $\neg F'$ with a $\mathcal{DNJ}$ J_{dn}. We select from each $\mathcal{DPJ}$ J_{dp} of F' one fact G and add $\sim G$ to J_{dn}. If J_{dp} contains a false undefined fact or a negative defined fact whose negation has a strong $\mathcal{J}$, then we select such a fact as G.

Assume J_{dp} does not contain such a fact. Assume moreover that for each positive defined fact F'' in J_{dp}, there exists a $\mathcal{PPJ}$ of F'' with all leaves at least unknown and no positive loops and all negative defined leaves have a negation with supported value at most unknown. By concatenating all these $\mathcal{PPJ}$ s with J_{dp} we obtain a $\mathcal{PPJ}$ for F' which contradicts condition (iv). Hence, there exists at least one positive defined fact which does not satisfy the assumption. Select such a fact as G.

The fixpoint of (J_i) is a $\mathcal{PNJ}$ for $\neg F$, and all undefined leaves are true, all defined leaves have a strong $\mathcal{J}$. Hence $\neg F$ has a strong $\mathcal{J}$, which is the desired contradiction.

So, at this point we have that for each fact F for which $SV_J(I, F) \leq \mathbf{u}$, $\sim F$ has a loop-free $\mathcal{PJ}$ such that all undefined leaves are at least unknown and for all defined leaves F', $SV_J(I, F') \leq \mathbf{u}$. Moreover, each defined fact occurs at least at depth 1 (since F and the defined leaves in its partial justification have a different sign). As indicated before we can now construct a monotonically increasing sequence (J_i) of open justifications for F, such that all undefined leaves are at least unknown, all defined leaves occur at least at depth i and moreover, the defined leaves of J_{i+1} have a different sign than the defined leaves of J_i. The sequence (J_i) converges to a justification for F, which comprises only leaves which are at least unknown. No positive loop

can occur since all partial justifications are loop-free and signs of defined leaves of J_i and J_{i+1} are different. Hence the justification is at least weak.

□

Since a justification comprises a partial justification which comprises a direct justification, one expects a relationship between the corresponding semantics.

Theorem 4.3.2 *A justified model of a program is a partially justified model. A partially justified model is a directly justified model.*

Proof Let I be a justified model. It suffices to show that for each positive or negative fact F: $SV_{PJ}(I,F) = SV_J(I,F)$. Consider a maximal justification J of F. Clearly, J contains a partial justification J_p for F as a subtree. We show that $val_I(J) \leq val_I(J_p)$. First, if $val_I(J)$ is $\mathbf{f}$, then nothing has to be proved. Second, assume that $val_I(J)$ is $\mathbf{t}$. All undefined leaves of J and J_p are true. For all defined leaves of J_p, J comprises a strong justification, so they must be true. J and therefore J_p do not contain a positive loop. Hence, J_p is strong.

Third assume $val_I(J)$ is $\mathbf{u}$. Again all undefined leaves of J_p are true and J_p does not contain a positive loop. All defined leaves of J_p must be at least unknown since J contains for each of them at least a weak justification. Hence, $val_I(J_p)$ is at least $\mathbf{u}$.

This proves that for each fact F: $SV_{PJ}(I,F) \geq SV_J(I,F)$. Assume that for some fact $SV_{PJ}(I,F) > SV_J(I,F)$. Then by theorem 4.3.1, $SV_{PJ}(I,\sim F) < SV_J(I,\sim F)$. This is a contradiction.

In exactly the same way it can be proved that a partially justified model is a directly justified model. □

An important consequence of this theorem is that wrt entailment, a logic consequence wrt $\mathcal{DJS}$ is a logic consequence wrt $\mathcal{PJS}$. A logic consequence wrt $\mathcal{PJS}$ is a logic consequence wrt $\mathcal{JS}$.

Theorem 4.3.3 *Let $\mathcal{T}$ be a theory, F some FOL formula both based on $\mathcal{L}$.*
(a) If $\mathcal{T} \models_{\mathcal{DJS}} F$ then $\mathcal{T} \models_{\mathcal{PJS}} F$
(b) If $\mathcal{T} \models_{\mathcal{PJS}} F$ then $\mathcal{T} \models_{\mathcal{JS}} F$

The proof is trivial. The theorem is important: it guarantees that a deductive or abductive procedure which is sound wrt $\mathcal{DJS}$ is also sound wrt $\mathcal{PJS}$ and $\mathcal{JS}$[3]. If it is sound wrt $\mathcal{PJS}$ it is sound wrt $\mathcal{JS}$. Given that $\mathcal{DJS}$ is equivalent with completion semantics (as was announced in the introduction and will be proved

[3] A model generation procedure sound wrt $\mathcal{DJS}$ is obviously not necessarily sound wrt $\mathcal{PJS}$ and $\mathcal{JS}$.

in section 4.4), the theorem implies also that the completion of a logic program can always be used as a sound first order logic approximation of the meaning of a program under $\mathcal{PJS}$ or $\mathcal{JS}$.

The next theorem asserts that each logic program is consistent wrt ($\mathcal{DJS}$) ($\mathcal{PJS}$) ($\mathcal{JS}$). Recall from section 2.1 that an *incomplete interpretation* for some subset of the predicates of $\mathcal{L}$ is a partial interpretation of which the truth function is defined only for the specified predicates.

Theorem 4.3.4 *Given is a logic program $\mathcal{T}_d$ and an incomplete interpretation I for the undefined predicates of $\mathcal{L}$ only.*

(a) There exists a directly justified model of $\mathcal{T}_d$ extending I.

(b) There exists a partially justified model of $\mathcal{T}_d$ extending I.

(c) There exists a unique justified model of $\mathcal{T}_d$ extending I.

Proof By theorem 4.3.2, (a) and (b) follow trivially from (c).

Let I be an incomplete interpretation for the undefined predicates. We define the interpretations of the defined predicates in the following way. Consider the set of all justifications for all positive defined facts in I. Observe that all leaves of a justification contain undefined facts. Therefore, given I, the value of a justification is completely determined. We extend $\mathcal{H}_I$ by defining for any defined simple positive fact F:

$$\mathcal{H}_I(F) = SV_J(I, F)$$

By definition of $\mathcal{H}_I$, the extension of I is a justified model. Moreover, since the values of the justifications are completely determined by I, I can be extended in only one way. □

Contrary to well-founded semantics, a logic program will have a class of justified models, one for each incomplete interpretation.

Another theorem gives a relation between a logic program P interpreted as a program and as a set of FOL axioms. It says that a directly justified model of a definition set P is a model of P interpreted as a set of FOL axioms.

Theorem 4.3.5 *Let P be a program and M a (directly justified)(partially justified)(justified) model of P. M is a model of P interpreted as a set of FOL axioms.*

Proof By theorem 4.3.2, it suffices to prove this for $\mathcal{DJS}$. To prove is that for any ground instance F :- $L_1, \ldots, L_n$, $\mathcal{H}_M(F) \geq \mathcal{H}_M(L_1 \wedge \ldots \wedge L_n)$. This is because $\mathcal{H}_M(F) = SV_{DJ}(\tilde{M}(F) = max(\{val_M(J_d) | J_d \text{ is a } \mathcal{DJ} \text{ of } \tilde{M}(F)\}) \geq \mathcal{H}_M(L_1 \wedge \ldots \wedge L_n)$.

□

The reverse direction is not true. For example, the set of FOL axioms $\{p \text{:-} \Box\}$ has a model in which p is true.

After having established relationships between the three types of semantics, we now investigate the differences between them. We will do so by taking the strongest concept of justification ($\mathcal{J}$) and investigating it in the context of the three types of semantics.

Lemma 4.3.2 *(a) Let M be a directly justified model.*

- *A true fact has a justification with only true nodes.*
- *An unknown fact has at least one justification with only true or unknown nodes. Each such justification contains an infinite branch of unknown facts.*
- *For a false fact, each justification contains a branch with false facts.*

(b) Let M be a partially justified model.

- *A true defined fact F has a strong or weak justification.*
- *An unknown fact F has a weak justification and no strong justification.*
- *A false fact has only false or weak justifications.*

A true fact in a directly justified model may have a false justification, which nevertheless contains only true facts. For example, consider the theory

$$\{q \text{:-} \blacksquare; \quad p \text{:-} q, p\}$$

The interpretation $\{q^{\mathbf{t}}, p^{\mathbf{t}}\}$ is a directly justified model. p has a false justification with only true leaves and an positive loop over p. On the other hand $\neg p$ is false but has a strong justification with a negative loop over $\neg p$, but this branch contains only false facts, as indicated by the lemma.

A true fact in a partially justified model may have only a weak justification. Consider the theory with definitions $p \text{:-} \neg q; q \text{:-} \neg p$. The interpretation $\{p^{\mathbf{t}}, q^{\mathbf{f}}\}$ is a partially justified model. p has only a weak justification but all facts in it are true.

Proof of lemma 4.3.2

(a) In a directly justified model M, each true or unknown simple fact F has a $\mathcal{DJ}$ with value $\mathcal{H}_M(F)$. This allows to build a monotonically increasing sequence (J_i) of open justifications with root F, such that for each J_i, all nodes have a truth value $\geq \mathcal{H}_M(F)$ and all open leaves occur at depth i. The fixpoint of this sequence contains only nodes with truth value $\geq \mathcal{H}_M(F)$ and no open leaves.

Assume F and $\sim F$ are unknown. Hence, they have both a justification, say J_1 and J_2, in which all nodes are at least unknown. By lemma 4.3.1, there exists a branch B_1 in J_1 such that $\sim B_1$ occurs in J_2. This branch necessarily contains only unknown facts, and since a finite branch terminates in a true or false fact, it must be an infinite branch.

Each false fact F is the negation of a true fact who has a justification with only true facts. Hence, by lemma 4.3.1, each justification of F must have a branch of false facts.

(b) The proof of (b) is very similar to (a). In a partially justified model, each true or unknown has a strong or weak partial justification. Using this information, it is possible to construct a justification which is at least weak. If F is unknown then so is $\sim F$. Since F and $\sim F$ have at least weak justifications and because of theorem 4.3.1, it follows that F has a weak but no strong justification. A fact which is false cannot have a strong justification, otherwise $\sim F$ could have only false justifications.

□

Proposition 4.3.1 *(a) Let M be a directly justified model. A fact with a strong justification without infinite branches is true.*

(b) Let M be a partially justified model. A fact with a strong justification is true.

Proof (a) Let M be a directly justified model. If a fact F has a strong justification J of finite depth, then by lemma 4.3.1, each justification of $\sim F$ has a finite branch with a false fact. Since by lemma 4.3.2, a true or unknown fact has a justification without false facts, $\sim F$ must be false. Hence, F is true.

(b) is a direct consequence of lemma 4.3.2: the only facts which can have strong justifications are true facts.

□

(b) has a consequence which is an extension of a result by [Prz90], that the well-founded model of program is the F-weakest stationary model. A stationary model is a 3-valued version of a stable model. The following definition for *F-weaker* is an extension of a definition for Herbrand interpretations given in [Kun89]:

Definition 4.3.1 *An interpretation I_1 is F-weaker than an interpretation I_2 if they share the same pre-interpretation and for each positive fact F which is true or false according to I_1, $\mathcal{H}_{I_1}(F) = \mathcal{H}_{I_2}(F)$.*

In other words, for all facts where I_1 differs from I_2, this can only be by having more unknown facts. I_1 contains less information than I_2.

Lemma 4.3.3 *(a) Let an interpretation I_1 be F-weaker than an interpretation I_2. For each closed domain formula F which is true or false according to I_1, $\mathcal{H}_{I_1}(F) = \mathcal{H}_{I_2}(F)$.*

(b) Let M' be a model of a theory $\mathcal{T}_c$ of FOL axioms. If an interpretation M is F-weaker than M', then M is a model of $\mathcal{T}_c$.

Proof (a) can be easily verified by induction on the structure of the domain axioms. (b) is a direct consequence of (a). Indeed, for each FOL axiom F of $\mathcal{T}_c$, since $\mathcal{H}_{M'}(F) = \mathbf{t}$ or $\mathbf{u}$, $\mathcal{H}_M(F)$ cannot be false. □

Theorem 4.3.6 *Let $\mathcal{T}$ be a theory consisting of program $\mathcal{T}_d$ and FOL axioms $\mathcal{T}_c$. For any partially justified model M, there exists a unique partially justified model M' minimally F-weaker than M. M' is a justified model. If I is the incomplete interpretation obtained by restricting M to the undefined predicates, then M' is the unique justified model extending I.*

Proof Let M be a partially justified model. Observe that because the interpretation of an undefined predicated is two-valued, any F-weaker partially justified model necessarily has the same interpretation for the undefined predicates. Since the value of a justification depends only on the value of the undefined facts in the leaves and the type of loops occurring in it, in all F-weaker partially justified models, each justification for a defined fact has the same value as in M. By proposition 4.3.1 (b), each fact with a strong justification in M is true in M and in any F-weaker partially justified model. Each fact with only false justifications is the negation of a fact with a strong justification (theorem 4.3.1), which must be true in M and in all F-weaker partially justified models. The fact itself must then be false in all these models.

By theorem 4.3.4 there exists a unique justified model M' of $\mathcal{T}_d$ which extends the restriction of M to the undefined predicates. By theorem 4.3.2, M' is a partially justified model of $\mathcal{T}_d$. By the reasoning in the previous paragraph, M' is the F-weakest partially justified model of $\mathcal{T}_d$. That M' is a model of $\mathcal{T}_c$ follows from lemma 4.3.3. □

The theorem says that for a partially justified model, there exists a unique F-weaker justified model, which may be strong or weak. The following example gives a strong partially justified model with only a weak F-weaker justified model.

$$\{p \text{ :- } \neg q; q \text{ :- } \neg p; p \vee \neg p\}$$

The interpretation $\{p^{\mathbf{f}}, q^{\mathbf{t}}\}$ is a strong partially justified model. The Justified model is $\{p^{\mathbf{u}}, q^{\mathbf{u}}\}$ which is only a weak model since $\mathcal{H}_I(p \vee \neg p) = \mathbf{u}$.

Theorem 4.3.3 proved that entailment wrt $\mathcal{PJS}$ is subsumed by entailment wrt $\mathcal{JS}$. A direct consequence of the above theorem is that with respect to entailment,

3-valued partially justified model semantics and 3-valued justified model semantics are equivalent.

Theorem 4.3.7 *Let $\mathcal{T}$ be a theory. For any closed formula F:*

$$\mathcal{T} \models_{\mathcal{PJS}} F \Leftrightarrow \mathcal{T} \models_{\mathcal{JS}} F$$

Proof By theorem 4.3.3, it holds that

$$\mathcal{T} \models_{\mathcal{PJS}} F \Rightarrow \mathcal{T} \models_{\mathcal{JS}} F$$

So assume $\mathcal{T} \models_{\mathcal{JS}} F$. For any partially justified model M, there is a unique F-weaker justified model M'. It follows that $\mathcal{H}_{M'}(F) = \mathbf{t}$. Lemma 4.3.3(a) implies that $\mathcal{H}_M(F) = \mathbf{t}$. Hence $\mathcal{T} \models_{\mathcal{PJS}} F$.

□

Another consequence is:

Proposition 4.3.2 *Let $\mathcal{T}$ be a theory with only two-valued justified models. Every partially justified model of $\mathcal{T}$ is a justified model.*

Proof For any partially justified model M of $\mathcal{T}$, there exists an F-weaker justified model M' of $\mathcal{T}$. Since M' is two-valued, M and M' are necessarily identical. □

Proposition 4.3.3 *For hierarchical logic programs, $\mathcal{DJS}$ $\mathcal{PJS}$and $\mathcal{JS}$ coincide and are 2-valued. For definite programs, $\mathcal{PJS}$ and $\mathcal{JS}$ coincide and are 2-valued.*

Proof Justifications in hierarchical programs and in definite programs cannot contain loops over negation. Hence, justified models are always 2-valued and partially justified models are justified models.

All justifications in a hierarchical program have finite depth, and hence, all justifications are either strong or false. By proposition 4.3.1(a), a fact F with a strong justification in a directly justified model M is true. A fact F with only false justifications in M must be false: by theorem 4.3.1, $\neg F$ has a strong justification and must be true. We find that for all facts F, $\mathcal{H}_M(F) = SV_J(M, F)$ and M is a justified model.

□

4.4 Direct Justification Semantics versus completion semantics.

As argued in section 4.1, direct justifications are underlying to completion semantics. Clark's completion semantics [Cla78] was the first semantics for complete

normal logic programs with negation. According to this semantics, the meaning of a program P is given by (the 2-valued models of) a classical theory $comp(P)$ which consists of the theory of Free Equality ($FEQ(\mathcal{L})$), also called Clark equality and the set of completed definitions of the predicates. Since then, completion semantics have been extended several times: [Fit85] and [Kun89] proposed 3-valued completion semantics. [CTT91] proposed a 2-valued completion semantics for incomplete logic programs. The completion of an incomplete program contains only completed definitions for the defined predicates.

In [Fit85] and [Kun89], the classical equivalence operator $\leftrightarrow$ is replaced by a new operator $\Leftrightarrow$. The truth function associated to some interpretation I is extended for $\Leftrightarrow$ as follows:

$$\mathcal{H}_I(E_1 \Leftrightarrow E_2) = \mathbf{t} \text{ iff } \mathcal{H}_I(E_1) = \mathcal{H}_I(E_2)$$
$$\mathcal{H}_I(E_1 \Leftrightarrow E_2) = \mathbf{f} \text{ iff } \mathcal{H}_I(E_1) \neq \mathcal{H}_I(E_2)$$

For two-valued interpretations, the meaning of the normal logical equivalence $\leftrightarrow$ and $\Leftrightarrow$ is the same.

In order to show the relationship with all these semantics in an economic way, we will first define a completion semantics which can be seen as the least upper-bound of the current completion semantics. Our completion semantics is for (possibly) incomplete general logic programs, is 3-valued and allows non-Herbrand interpretations.

Let P be a complete or incomplete general logic program. We define $comp_3(P)$ as the union of $FEQ(\mathcal{L})$ and the set of completed definitions of defined predicates, in which " $\Leftrightarrow$ " is substituted for "$\leftrightarrow$ ". For any theory $\mathcal{T}$ consisting of a logic program $\mathcal{T}_d$ and FOL axioms $\mathcal{T}_c$, we define that an interpretation M of $\mathcal{L}$ is a model of a theory $\mathcal{T}$ wrt 3-valued completion semantics iff M is a model of the set of FOL axioms $comp_3(\mathcal{T}_d) + \mathcal{T}_c$ which is 2-valued on the undefined predicates.

Theorem 4.4.1 *Let $\mathcal{T}$ be a theory with normal logic program P (with finite definitions) and FOL axioms $\mathcal{T}_c$. $\mathcal{T}_c$ includes $FEQ(\mathcal{L})$. Let M be an interpretation of $\mathcal{L}$.*

(a) M is a directly justified model of $P + FEQ(\mathcal{L})$ iff M is 3-valued model of $comp_3(P)$.

(b) M is a directly justified model of $\mathcal{T}$ iff M is a model of $\mathcal{T}$ wrt 3-valued completion semantics.

The theorem explicitly requires that P should not contain infinite definitions. Obviously, the completion definition of a predicate with infinite definition is not well-defined since the completed definition of a predicate with infinite number of clauses is not defined. On the other hand, the direct justification semantics for infinite definitions is well-defined. We will use this later when we consider abductive solutions Δ as possibly infinite definitions for the undefined predicates.

Proof (a) Models of $comp_3(P)$ and directly justified models of $P + FEQ(\mathcal{L})$ are models of $FEQ(\mathcal{L})$. Hence, it suffices to show that a directly justified model of P is a model of $comp_3(P)$ without $FEQ(\mathcal{L})$ and vice versa. Let M be an interpretation and let $p(x_1, \ldots, x_n)$ be a simple defined fact of I. Assume p/n has a completed definition of the form $p(X_1, \ldots, X_n) \Leftrightarrow \phi_1 \vee \ldots \vee \phi_k)$. Let V be the variable assignment $\{X_1/x_1, .., X_n/x_n\}$. We must show that $SV_{DJ}(M, p(x_1, \ldots, x_n)) = \mathcal{H}_M(V(\phi_1 \vee \ldots \vee \phi_k))$.

By definition of $\mathcal{H}_M$, we have:

$$\mathcal{H}_M(V(\phi_1 \vee \ldots \vee \phi_k)) = max\{\mathcal{H}_M(V(\phi_1)), \ldots, \mathcal{H}_I(V(\phi_k))\}$$

For each ϕ_i, there exists a rule of P of the form $p(t_1, .., t_n) \text{:-} B$ such that ϕ_i is $\exists Y_1, .., Y_m : X_1 = t_1 \wedge .. \wedge X_n = t_n \wedge B$. The only places where the X_j's occur in $V(\phi_i)$ is precisely in the equalities $X_j = t_j$. Therefore, $V(\phi_i)$ is of the form $\exists Y_1, .., Y_m : x_1 = t_1 \wedge .. \wedge x_n = t_n \wedge B$. $Y_1, .., Y_m$ are precisely the variables of the rule $p(t_1, .., t_n) \text{:-} B$. Or, any variable assignment V' for the variables $Y_1, .., Y_m$ corresponds with a ground instance of the rule. We have that:

$$\mathcal{H}_M(V(\phi_1 \vee \ldots \vee \phi_k)) = max\{\mathcal{H}_M(V'(x_1 = t_1 \wedge .. \wedge x_n = t_n \wedge B)) | \\ V'(p(t_1, .., t_n) \text{:-} B) \text{ is a ground instance of a rule of } P\}$$

Because the interpretation of "=" is the identity relation, $V'(x_1 = t_1 \wedge .. \wedge x_n = t_n)$ is not false iff for each $1 \leq j \leq n$, $\tilde{M}(V'(t_j)) = x_j$. The other instances may be dropped from the set in the right of the equation. The instances which satisfy the condition, have the property that the conjunction of equalities is true and that $\tilde{M}(V'(p(t_1, .., t_n))) = p(x_1, .., x_n)$. Or, such an instance corresponds to a direct positive justification J for $p(x_1, .., x_n)$. Moreover, $val_M(J) = \mathcal{H}_M(V'(B)) = \mathcal{H}_M(V'(x_1 = t_1 \wedge .. \wedge x_n = t_n \wedge B))$. We find that $\mathcal{H}_M(V(\phi_1 \vee \ldots \vee \phi_k))$ is equal to

$$max\{val_M(J) | J \text{ is a direct justification of } p(x_1, .., x_n)\}$$

which is equal to $SV_{DJ}(M, p(x_1, .., x_n))$. This proves the theorem[4].

The proof of (b) follows easily from (a). There is a subtle point: the reasoning is based on monotonicity of FOL semantics. In general a model of a theory $\mathcal{T}_1 + \mathcal{T}_2$ in a nonmonotonic logic is not necessarily a model of $\mathcal{T}_1$ and of $\mathcal{T}_2$.

[4]The theorem would not hold without the restriction that "=" is interpreted by identity. Consider the program $P = \{p(a) \text{:-} \blacksquare\}$. The interpretation $(D = \{x, y\}, \{a \rightarrow x\}, \{x = y^{\mathrm{t}}, y = x^{\mathrm{t}}, x = x^{\mathrm{t}}, y = y^{\mathrm{t}}, p(x)^{\mathrm{t}}, p(y)^{\mathrm{t}}\})$ is a model of $comp_3$(P) but is not a directly justified model of P because $p(y)^{\mathrm{t}}$ has no direct justification. A solution would be to consider the homogeneous form of the rules : $p(X) \text{:-} X = a$. For such programs, the theorem holds even when "=" is not interpreted by identity.

Below, we make a careful reasoning. Consider any interpretation M. Assume that M is a directly justified model of $P + \mathcal{T}_c$. M is a directly justified model of P and a model of $\mathcal{T}_c$, a model of $FEQ(\mathcal{L})$ (by monotonicity of FOL) and a directly justified model of $P + FEQ(\mathcal{L})$. Now we can apply (a), which guarantees that M is a model of $comp_3(P)$. Again by the monotonicity of FOL, we find that M is a model of $\mathcal{T} = comp_3(P) + \mathcal{T}_c$. The other direction is completely analogous.

□

Since all current completion semantics can be considered as instances of the completion semantics defined above, the following theorem is a direct consequence.

Theorem 4.4.2 *Let P be a logic program (with finite definitions), M an interpretation of $\mathcal{L}$. First, assume P is a complete logic program.*

M is a model of P according to Clark's completion semantics [Cla78] and M interprets "=" as identity iff M is a 2-valued directly justified model of $P + FEQ(\mathcal{L})$. [5]

M is a model of P according to Fittings completion semantics [Fit85] iff M is a directly justified Herbrand model of $P + FEQ(\mathcal{L})$.

M is a model of P according to Kunen's completion semantics [Kun89] iff M is a directly justified model of $P + FEQ(\mathcal{L})$.

Second, assume P is an incomplete logic program. M is a model of P according to the completion semantics of Console, Theseider Dupre and Torasso [CTT91] and M interprets "=" as identity iff M is a 2-valued directly justified model of $P + FEQ(\mathcal{L})$.

4.5 Relationship with stable and well-founded semantics

As shown first by [Fag90], the notion of justification found in stable models [GL88] is the partial justification. Stable semantics were developed for complete logic programs and are based on 2-valued Herbrand interpretations. [Prz90] extended stable semantics to 3-valued semantics for complete logic programs, called stationary semantics. [KM90b] extended stable semantics to 2-valued semantics for incomplete logic programs with FOL axioms, and called it generalised stable semantics. [PAA91b] extended well-founded semantics for incomplete logic programs in a similar way.

Partial justification semantics is strictly more general than each of these semantics, due to the fact that partially justified models can be 3-valued non-Herbrand interpretations. The relationships between justification semantics and

[5] For an uncontracted model of Clark's completion semantics, we have that its contraction over M("=") is a directly justified model of $P + FEQ(\mathcal{L})$.

well-founded, stable and stationary model semantics are given in two separate steps. In the first step we show that by restricting (partial) justification semantics to Herbrand interpretations, we obtain stable, stationary and well-founded models. In the second step we show how by adding FEQ and a strong domain closure axiom, we can restrict justification semantics such that all models are isomorphic with a stable, stationary or well-founded model.

We start by recalling definitions for stationary, stable and well-founded models, taken from [Prz90].

Given a Herbrand interpretation I, P_I is defined as the definite program obtained by taking all ground instances of rules of P and replacing the negative literals $\neg A$ by special symbols $\blacksquare$, $\blacksquare$ or $\square$ depending whether $\mathcal{H}_I(\neg A)$ is $\mathbf{t}$, $\mathbf{u}$ or $\mathbf{f}$ respectively. We define $\mathcal{H}_I(\blacksquare) = \mathbf{u}$ and $\sim\blacksquare = \blacksquare$.

Stationary models are defined via the notion of 3-valued least Herbrand model of the set of FOL axioms P_I. On the set of Herbrand interpretations, the relationship $\preceq$ can be defined as expected, namely $I_1 \preceq I_2 \Leftrightarrow \forall A \in HB(\mathcal{L}) : \mathcal{H}_{I_1}(A) \leq \mathcal{H}_{I_2}(A)$. $\preceq$ defines a partial order on the set of Herbrand interpretations. It was proven in [Prz90] that a definite program (allowing $\blacksquare$, $\blacksquare$ and $\square$) interpreted as a set of FOL axioms, has always a least Herbrand model.

Using these definitions, one can define stationary, stable and well-founded models as follows. Given some complete logic program P, a stationary model M of P is a Herbrand interpretation such that M is the least Herbrand model of P_M. A stable model is a 2-valued stationary model. The well-founded model is the unique F-weakest stationary model of P[6]. A generalised stable model for an incomplete logic program P and a set of FOL axioms is a model for the FOL axioms and a stable model of the complete logic program consisting of P and definitions for each of the undefined predicates. Each definition is a possibly infinite set of ground atomic rules. Similarly, a model of in incomplete logic program P wrt to the semantics of [PAA91b] is the well-founded model of the complete logic program consisting of P and definitions for each of the undefined predicates.

Theorem 4.5.1 *(a) A partially justified Herbrand model of a complete logic program is a stationary model and vice versa.*

(b) A 2-valued partially justified Herbrand model of a complete logic program is a stable model and vice versa.

(c) A 2-valued partially justified Herbrand model of an (incomplete) logic program is a generalised stable model and vice versa.

The proof of the theorem is based on the following lemma.

Lemma 4.5.1 *Let P be a complete definite program based on a language $\mathcal{L}$.*

The least Herbrand model of P interpreted as a set of FOL axioms is the unique justified and partially justified Herbrand model of P.

[6] This is not the original definition of well-founded model, but a theorem in [Prz90].

Proof By theorem 4.3.4, there is a unique justified Herbrand model M of P. By proposition 4.3.3, M is the unique partially justified model of P.

We show that M is the least Herbrand model of P wrt $\preceq$ where we interpret P here as a set of FOL axioms. By theorem 4.3.5, M is a model of P. To prove that M is the Least Herbrand model, it suffices to show that every positive simple fact F with a strong $\mathcal{PJ}$ J is true and every positive fact F with a weak $\mathcal{PJ}$ J is at least unknown in all Herbrand models which extend I. The proof is an easy induction on the depth n of J and is left to the reader.

□

Proof of theorem 4.5.1 The whole theorem is based on (a). Since a stable model is a two-valued stationary model, (b) follows directly from (a). As a generalised stable model is defined as a stable model of an incomplete program augmented with some (possibly infinite) definition for the undefined predicates, also (c) directly follows from (a).

The proof of (a) follows easily from lemma 4.5.1. Let M be a partially justified Herbrand model of P. Consider P_M. Clearly, each $\mathcal{PJ}$ for a positive fact in P can be turned into a $\mathcal{PJ}$ for the same fact in P_M by replacing negative facts in the leaves by the predicates ■, ⬓ or □ depending on the truth value of the negative facts. This transformation preserves the value of the $\mathcal{PJ}$. Hence M is a partially justified model of P_M, and by lemma 4.5.1, it is the least Herbrand model of P_M. Hence M is a stationary model of P.

The other direction is completely similar: since M is the least Herbrand model, M is the partially justified model of P_M. Due to the correspondence of $\mathcal{PJ}$'s in P and in P_M, it follows that M is a partially justified model of P.

□

Theorem 4.5.2 *(a) The justified Herbrand model of a complete logic program is the well-founded model.*

(b) A justified Herbrand model of an incomplete logic program is a model wrt the semantics of [PAA91b] and vice versa.

Proof (a) and (b) follow from theorem 4.5.1(a) and the fact that the F-least stationary model is the well-founded model and the F-least partially justified model is the justified model (theorem 4.3.6). □

We have found that by adding *FEQ* and a strong domain closure axiom (DCA), partial justification semantics coincides with stable, generalised stable and stationary semantics and justification semantics is equivalent with well-founded semantics. The DCA expresses that all elements in the domain correspond to terms. A

well-known weak approximation of the DCA is the FOL formula:

$$\forall X : (\exists \overline{Y}_1 : X = f_1(\overline{Y}_1)) \vee ... \vee (\exists \overline{Y}_m : X = f_n(\overline{Y}_n))$$

where $f_1, .., f_n$ are all the functors and constants of $\mathcal{L}$[Rei78a] [Llo87]. The weak DCA is not sufficient for languages which contain functors with arity > 0. As a well-known example, take the language with one functor f/1. Since it does not contain constants, the associated Herbrand universe is empty. However, here is a nonempty model of $\forall X : \exists Y : X = f(Y)$: take the integer numbers and interpret f/1 as $X + 1$. However, interestingly, under (partial) justification semantics the DCA can be expressed using a simple complete logic program and one FOL axiom.

Definition 4.5.1 *The Strong Domain Closure Axiom (*SDCA*) for a language $\mathcal{L}$ consists of a definition for a special predicate $\mathcal{U}/1$ and one FOL axiom. For each functor f/n $(n > 0)$ of $\mathcal{L}$, the definition contains the definite clause :*

$$\mathcal{U}(f(X1, .., X_n)) \text{ :-} \mathcal{U}(X_1), .., \mathcal{U}(X_n)$$

For each constant c of $\mathcal{L}$, the definition contains:

$$\mathcal{U}(c) \text{ :-} \blacksquare$$

The FOL axiom is:

$$\forall X : \mathcal{U}(X)$$

Note that the program consists only of definite clauses. Therefore, a partial justification is a justification and a partially justified model is a justified model. In a justified model I of the $SDCA$, $\mathcal{U}(x)$ has a finite justification for every domain element x. With this finite justification, a finite term t can be associated such that $\tilde{I}(t) = x$. Therefore each domain element is in the image of the Herbrand universe. Adding FEQ avoids that two terms are mapped on the same domain element. The combination of the $SDCA$ and FEQ allows only models isomorphic with some Herbrand interpretation. This is proved in the following theorem.

Theorem 4.5.3 *Any (partially) justified model of* SDCA *and* FEQ *is isomorphic with a Herbrand interpretation.*

Proof Let M be a justified model of $SDCA$ and FEQ. Remember that M can be extended as a mapping $\tilde{M}$ of all domain terms to D. We define h as the restriction of $\tilde{M}$ to HU and prove that h is a one-to-one correspondence from HU to D.

Since M is a model of FEQ, two distinct terms necessarily are mapped on different domain elements. That each domain element is in the image of h can be seen as follows. It holds that M is a justified model of $SDCA$. Hence, any domain element x has a justification J_x for $\mathcal{U}(x)$. Now, we show

that with J_x a unique term t_x can be associated, moreover $h(t_x) = x$. We define this association by induction: if the depth of J_x is 1, then it is by using a rule $\mathcal{U}(c)$:- ■, where c is a constant. By definition, $M(c) = h(c) = x$. Assume that J_x is of depth n, and the theorem holds for terms of depth $n-1$ ($n > 0$). Then the top of J_x is an application of an instance of some rule $\mathcal{U}(f(X_1, .., X_n))$:- $\mathcal{U}(X_1), .., \mathcal{U}(X_m)$ with $m > 0$. Hence, there is a variable assignment V of domain elements to variables $X_1, .., X_m$ such that $x = \tilde{M}(f(V(X_1), .., V(X_m)))$. Moreover, $\mathcal{U}(V(X_i))$ has a justification of depth at most $n-1$ for each i. By the induction hypothesis, there exist terms $t_1, .., t_m$ such that $h(t_i) = V(X_i)$. Hence, $h(f(t_1, .., t_m)) = \tilde{M}(f(t_1, .., t_m) = x$.

This concludes the proof that h is a one-to-one correspondence between HU and D. Using h, we can define the isomorphic Herbrand interpretation I by defining for all facts $p(t_1, .., t_n)$, $\mathcal{H}_I(p(t_1, .., t_n)) = \mathcal{H}_M(p(t_1, .., t_n))$. h is obviously an isomorphism between M and I. □

It can easily be verified that the *SDCA* under $\mathcal{DJS}$ is equivalent with the weak domain closure assumption in FOL. Hence, a model of a theory including *FEQ* and *SDCA* under $\mathcal{DJS}$ is not necessarily equivalent with a Herbrand interpretation.

Finally we obtain the following theorems.

Theorem 4.5.4 *(a) Stable semantics [GL88] for a complete logic program P is equivalent with 2-valued partial justification semantics for P augmented with* SDCA *and* FEQ.

(b) Generalised stable semantics [KM90b] for an incomplete program P is equivalent with 2-valued partial justification semantics for P augmented with SDCA *and* FEQ.

(c) Stationary semantics [Prz90] for a complete logic program P is equivalent with partial justification semantics for P augmented with SDCA *and* FEQ.

Theorem 4.5.5 *(a) Well-founded model semantics for a complete logic program P is equivalent with justification semantics for P augmented with* SDCA *and* FEQ.

(b) The semantics of [PAA91b] for incomplete logic programs P is equivalent with justification semantics for P augmented with SDCA *and* FEQ.

In the theorems, equivalence means that any model according to one semantics is a model or is isomorphic to some model according to the other semantics.

Both theorems are direct implications of theorems 4.5.1, 4.5.2 and 4.5.3 and the fact that a Herbrand interpretation of P satisfies *SDCA* and *FEQ*.

4.6 On duality of abduction and model generation

In chapter 3.8, we showed a duality between abduction and model generation. Abductive solutions of a definite logic programs were shown to correspond to models of the dual theory *only-if*(P). Here we prove some related properties which extend this to normal programs. Below we interpret a set Δ of ground undefined atoms based on $\mathcal{L}$ as a set of definitions of undefined predicates (mapping an atom $A \in \Delta$ to a clause A:-■). With Δ, an incomplete Herbrand interpretation I_Δ for the undefined predicates is associated as follows $\mathcal{H}_{I_\Delta}(A) = \mathbf{t}$ if $A \in \Delta$ and $\mathcal{H}_{I_\Delta}(A) = \mathbf{f}$ if $A \notin \Delta$.

Theorem 4.6.1 (Weak Duality) *Given a logic program P, a query Q and a set Δ of (possibly infinite) definitions for the undefined predicates of P consisting of ground atoms.*

(a) $P + \Delta \models \neg Q$ wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$ implies that $P + \neg Q$ and $P + \Delta$ are consistent wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$. I_Δ can be extended to a model of $P + \neg Q$.

(b) $P + \mathrm{FEQ} + \Delta \models \neg Q$ wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$ implies that $P + \mathrm{FEQ} + \neg Q$ and $P + \mathrm{FEQ} + \Delta$ are consistent wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$. I_Δ can be extended to a model of $P + \mathrm{FEQ} + \neg Q$.

(c) $P + \mathrm{FEQ} + \mathrm{SDCA} + \Delta \models \neg Q$ wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$ implies that $P + \mathrm{FEQ} + \mathrm{SDCA} + \neg Q$ and $P + \mathrm{FEQ} + \mathrm{SDCA} + \Delta$ are consistent wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$. I_Δ can be extended to a model of $P + \mathrm{FEQ} + \mathrm{SDCA} + \neg Q$.

Proof (a) and (b) can be proven like (c). We prove only (c). By theorem 4.3.4 the theory $P + FEQ + SDCA + \Delta$ has a Herbrand model M wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$. Since $P + FEQ + SDCA + \Delta \models \neg Q, \neg Q$ is true in M. It is straightforward that M is a model of the incomplete program P. Note that a necessary condition for this is that the interpretations of the undefined predicates should be two-valued. This is the case since Δ is a hierarchical program (proposition 4.3.3). Hence, $P + FEQ + SDCA + \neg Q$ and $P + FEQ + SDCA + \Delta$ are consistent wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$. Clearly M is an extension of I_Δ. Hence, I_Δ can be extended to a model of $P + FEQ + SDCA + \neg Q$. □

Therefore, any sound abductive procedure which returns a solution Δ for a query Q, proves not only $P + \Delta \models \neg Q$ but also the consistency of $P + \Delta(+\neg Q)$ and performs a form of partial model generation generating a partial model I_Δ. The implicit consistency proof and partial model generation can be seen as the dual interpretation of the abductive derivation. A pleasant side-effect is that it simplifies the task of computing abductive solutions, since proving the consistency of the abductive solution with the program is not longer necessary. This consistency checking is necessary in other semantics and is costly (see [SI92] for the generalised stable semantics).

There is a stronger theorem which was announced in section 3.8.

Theorem 4.6.2 (Strong Duality) *Given a logic program P, a query Q and a set Δ of (possibly infinite) definitions for the undefined predicates of P consisting of ground atoms.*

$P + \Delta +$ SDCA + FEQ $\models_{\mathcal{JS}} \neg Q$ iff $P +$ SDCA + FEQ $+ \neg Q$ has a justified (Herbrand) model which extends I_Δ.

Proof The only-if direction was proved in theorem 4.6.1(c). It suffices to prove the if direction. Consider this justified Herbrand model M of $P + SDCA + FEQ + \neg Q$ which interprets the undefined predicates by I_Δ. Evidently it is a model of the complete program $P + \Delta$. Hence, it is a justified model of $P + \Delta + SDCA + FEQ$. However, by theorem 4.5.3 all justified models of $P + \Delta + SDCA + FEQ$ are isomorphic with M. Hence, in all justified models, $\neg Q$ holds. □

4.7 Negation as failure as abductive reasoning

By making explicit how true facts are constructed in different semantics, the framework shows that only in well-founded and justification semantics, a true fact never depends on itself, and therefore these semantics provide the best formalisations of the view of logic programs as sets of constructive definitions. However, the stable semantics for logic programs is supported by a competing view on logic programs which has been promoted in [EK89], [Dun91], [KM90c]. According to this view, negation as failure corresponds to a special form of abductive or default reasoning. The theory is based on maximal sets Δ of negative literals, such that $P + \Delta$ has a least Herbrand model. This least Herbrand model turns out to be a stable model. As an example, consider the following program:

$$P_1 = \{\ p \text{ :- } \neg q \\ q \text{ :- } \neg p \\ r \text{ :- } p \\ r \text{ :- } q\ \}$$

The set of abductive hypotheses $\Delta = \{\neg q\}$ is maximally consistent with P: it is consistent since q cannot be derived from $P + \Delta$. It is maximal in the sense that the addition of any negative literal to Δ causes Δ to be inconsistent with P. When Δ is added to the program, the stable model $\{p^{\mathbf{t}}, r^{\mathbf{t}}\}$ is obtained. Another maximal consistent set of abductive hypotheses is $\{\neg p\}$. The corresponding stable model is $\{q^{\mathbf{t}}, r^{\mathbf{t}}\}$.

The two views are not equivalent and lead to different programming styles. According to the view of programs as constructive definitions, the above example is simply a *bad* program, with unique well-founded/justified model $\{p^{\mathbf{u}}, q^{\mathbf{u}}, r^{\mathbf{u}}\}$.

Remember that we interpret **u** as *locally inconsistent*. Despite the difference between the two views, it is important to note that procedures developed for stable semantics or for partially justified semantics may be correctly applied for all *good* programs under the constructive definition view. The *good* programs or theories are those for which each model is contradiction-free. Formally, we call a theory *overall consistent* if all its justified models are two-valued. We have the following property:

Proposition 4.7.1 *An overall consistent theory has the property that all partially justified models are justified models.*

This property is a direct consequence of theorem 4.3.6.

Programmers whom adhere to the constructive definition view, can be assumed to write *overall consistent* theories. The above proposition implies that for overall consistent theories, any theorem prover or abductive procedure or what ever procedure which satisfies some correctness property wrt partial justification semantics, satisfies this property also wrt justification semantics. An important consequence is that procedures developed for stable semantics, stationary or partially justified semantics are acceptable for execution under justification semantics, under the provision that the user tries to avoid "bad" theories, i.e. theories which are not overall consistent.

The following theorem indicates some classes of overall consistent theories.

Theorem 4.7.1 *The following classes of theories are overall consistent:*

- *theories with definite logic programs*
- *theories with hierarchical logic programs*
- *theories with stratified logical programs*
- *theories with locally stratified programs and with* SDCA *and* FEQ
- *theories with acyclic programs and with* SDCA *and* FEQ

Proof An unknown fact in a partially justified model has a weak justification, i.e. a justification which contains a loop over negation. Such justifications do not exist for definite clauses. For hierarchical programs, justifications never contain an infinite branch. For stratified programs, any justification can only contain a finite number of switches over negation. A model of a theory containing *SDCA* and *FEQ* is isomorphic with a Herbrand model. In a Herbrand model of a locally stratified program, justifications never contain a loop over negation. The same reasoning holds for acyclic programs with *SDCA* and *FEQ*[7].

[7] A theory with a locally stratified or acyclic logic program without *SDCA* and *FEQ* is not necessarily overall consistent. Take the locally stratified program $\{p(a) \text{ :- } \neg p(b)\}$. The interpretation: $(D = \{x\}, \{a \rightarrow x, b \rightarrow x\}, \{p(x)^{\mathbf{u}}\})$ is a justified model.

□

4.8 On the nature of negation

A declarative characterisation of negation in classical logic is that a predicate p/n and its negation $\neg p/n$ represent *complementary concepts*: two concepts are complementary if one is true iff the other one is not true. This is reflected in the fact that the law of the excluded middle holds. In three-valued logics, the law of excluded middle does not hold. Therefore, in principle three-valued logics allow to represent two non-complementary concepts by a predicate and its negation. A standard example which exploits this phenomenon in well-founded semantics, has been given in e.g. [GL88], [VRS91], [Prz91]:

$winning(X)$:- $move(X,Y), \neg winning(Y)$
$move(a,b)$:- ■
$move(b,c)$:- ■
$move(d,d)$:- ■

Note that $winning(d)$ has a justification with loop over negation. This program has one 3-valued justified Herbrand model which interprets $winning/2$ as $\{winning(b)^{\mathbf{t}}, winning(d)^{\mathbf{u}}\}$. [VRS91] [Prz91] interpret $winning(d)^{\mathbf{u}}$ as a draw on d. Clearly, $winning/1$ and $\neg winning/1$ are not complementary since playing a draw is a third alternative. Here $\neg winning/1$ represents the concept of *loosing*, which is indeed not the complement of *winning*.

The above interpretation of $winning(d)^{\mathbf{u}}$ as a draw does not match with the interpretation of $\mathbf{u}$ as local inconsistency. A natural representation of the problem in the constructive definition view is by introducing $loosing/1$ and $draw/1$ predicates. Intuitively, X looses if no move from X exists or when a move from X to Y exists and Y is winning. X wins if a move from X to Y exists and Y is loosing. The problem is then represented by:

$loosing(X)$:- $\neg existsmove(X)$
$existsmove(X)$:- $move(X,Y)$
$loosing(X)$:- $move(X,Y), winning(Y)$
$winning(X)$:- $move(X,Y), loosing(Y)$
$draw(X)$:- $\neg loosing(X), \neg winning(X)$
$move(a,b)$:- ■
$move(b,c)$:- ■
$move(d,d)$:- ■

If in addition we add FEQ and $SDCA$, this program is a correct representation of the intended meaning. The justified Herbrand model is 2-valued and interprets $winning, loosing$ and $draw$ as $\{winning(b)^{\mathbf{t}}, loosing(a)^{\mathbf{t}}, loosing(c)^{\mathbf{t}}, draw(d)^{\mathbf{t}}\}$.

Both $winning(d)$ and $loosing(d)$ have only false justifications which contain the positive loop $\ldots \leftarrow winning(d) \leftarrow loosing(d) \leftarrow winning(d) \leftarrow \ldots . loosing(c)$ has the justification, shown in figure 4.5:

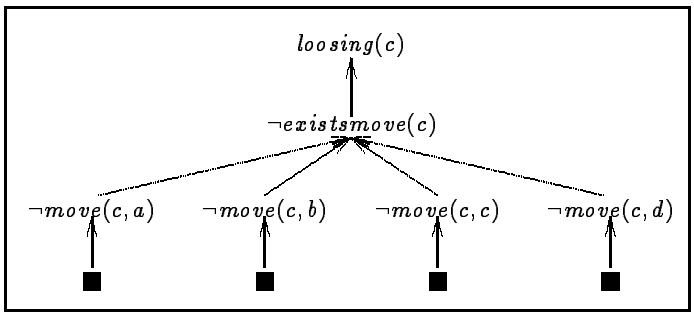

Figure 4.5: Justification of $loosing(c)$

This example illustrates that the constructive definition view in justification semantics can still be reconciled with the classical view on negation to represent complementary concepts, despite its 3-valued nature. This is due to **u**'s interpretation as *locally inconsistent.*

The interpretation of negation in logic programming is currently an issue subject to lively discussions, mainly due to work of [GL90a] on extended logic programming. This formalism allows two forms of negation: one is called *classical negation*, the other *negation as failure* or *negation by default.* The term *classical negation* used by [GL90a] does not appeal to what we called the classical view on negation (negation to represent complementary concepts). As a matter of fact, the law of excluded middle does not hold for the "classical negation" but it does hold for the negation by default in extended logic programming. Rather, their *classical negation* seems to refer to the meaning of negation in the logic program interpreted not as a logic program but as a set of FOL implications. The work of [GL90a] seems an attempt of integrating these two different interpretations of a logic program (as a program and as a set of FOL formulas) into one formalism. A comparison between the extended logic programming and incomplete logic programming is found in chapter 6.

4.9 Representing incomplete knowledge

We recall an example of section 4.7:

$$P_1 = \{\ p \text{:-} \neg q$$

q :- $\neg p$
r :- p
r :- q }

Note that this program under the stable semantics represents *incomplete knowledge*: it implies r and $(p \vee q) \wedge (\neg p \vee \neg q)$ without implying p or implying q. This use of mutual recursion over negation for representing incomplete knowledge in stable semantics occurs for example in [Dun92]. The well-founded and justified model of P_1 is $\{p^{\mathbf{u}}, q^{\mathbf{u}}, r^{\mathbf{u}}\}$. By interpreting $\mathbf{u}$ as *unknown*, the program does also represent incomplete knowledge on p and q. However, it is a poor way of representing incomplete knowledge: neither $(p \vee q) \wedge (\neg p \vee \neg q)$ nor r are implied. Moreover, P_1 conflicts with the constructive definition view.

The interpretation of $\mathbf{u}$ as unknown is only sensible in what we called a *knowledge state semantics* in section 2.2. In this type of semantics, a model represents what atoms are known to be true, what atoms are known to be false and what atoms have unknown truth value. At present, it seems that this is the dominant view on semantics in logic programming. In contrast, we adhere to the classical view on model semantics and view the justification semantics as a *possible state semantics*. In a possible state semantics, a model represents a possible state of the problem domain. Incomplete knowledge is represented by incomplete theories which have essentially different models. To represent incomplete knowledge, a possible state semantics is superior to a knowledge state semantics.

As an example, the incomplete knowledge in P_1 can be represented in justification semantics as follows. Since there is incomplete knowledge on p and q, it is natural to choose them as undefined predicates and to add the FOL axiom $(p \vee q) \wedge (\neg p \vee \neg q)$ to the theory. One gets the following theory:

$\mathcal{T}_1 = \{$ $(p \vee q) \wedge (\neg p \vee \neg q)$
r :- p
r :- q }

It is easy to see that $\mathcal{T}_1$ has precisely the same models and logical consequences under justification semantics as P_1 under stable model semantics. Moreover, this program is in full agreement with the constructive definition view. This use of undefined predicates and FOL axioms is an expressive way of representing incomplete knowledge since it comprises first order logic. The expressivity of first order logic for representing incomplete knowledge is widely accepted.

To represent incomplete knowledge is also our main motivation for allowing general interpretations in justification semantics instead of Herbrand interpretations. Intuitively, to have incomplete knowledge on what are the entities of some problem domain means that not all possible states of the problem domain have the same set of entities. In a possible state semantics, this is reflected by having models with different domains. Of course, in each model the domain should include the set of known objects, i.e. the elements of the Herbrand universe. However, other

elements may occur also. In Herbrand interpretation based semantics, the *SDCA* and *FEQ* are always satisfied. In all applications in which the domain is known to correspond to the Herbrand universe, this will result in a correct theory. However, when the knowledge engineer has incomplete knowledge about the domain of the intended interpretation, incorrect theorems may be provable from the logic program. As a trivial example, consider a language with predicate $p/1$ and constant a. The theory is:

$$P = \{p(a) \text{ :- } \blacksquare\}$$

In any Herbrand based semantics, P implies $\forall X : p(X)$. Assuming that the knowledge engineer has incomplete knowledge on the domain, this may be a wrong conclusion. It may well be that there is another element b in the intended domain and for which $p(b)$ does not hold. The only correct behaviour here is that neither $\forall X : p(X)$ nor $\exists X : \neg p(X)$ should be implied. This is precisely what happens under (partial) justification semantics and this is due to the fact that P has (general) models in which a is the only domain element and others in which there are additional domain elements. If the knowledge engineer at a later stage learns that a is the only element, then he can add the *SDCA*. $P + SDCA$ implies $\forall X : p(X)$. This way, justification semantics allows to represent the two situations correctly. In chapter 7, we give a natural example of a domain in which incomplete knowledge on the problem domain exists.

A remarkable observation is that the pure logic program formalism without FOL axioms is as expressive as with FOL axioms. Below we show that any set of FOL axioms can be transformed to an *elementary extension* consisting of an incomplete logic program and one simple FOL axiom $\neg false$ (see definition 2.1.4). This means that every model of the original theory (wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$) is the restriction of a model of the transformed theory (wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$), to the symbols of $\mathcal{L}$ and vice versa, a model of the latter theory restricted to the symbols of $\mathcal{L}$ is a model of the original theory.

The transformation was used before in [SK88] in the context of checking integrity constraints in databases. It is an extension of the well-known procedure in [LT84] by a pre-processing step, in which each FOL axiom F of the theory is taken and replaced by the rule $false \text{ :- } \neg F$. In addition the FOL axiom $\neg false$ is added to the theory. In the second step, the algorithm of [LT84] is applied on the definition of $false$.

Algorithm 4.9.1 *Given is a theory $\mathcal{T} = \mathcal{T}_d \cup \mathcal{T}_c$ based on $\mathcal{L}$. We assume without loss of generality that the formulas in $\mathcal{T}_c$ contain only logic operators $\neg$, $\wedge$, $\vee$, $\exists$ and $\forall$. The transformation proceeds as follows:*

step 1: Define $P = \{false \text{ :- } \neg F | F \in \mathcal{T}_c\}$

step 2: The algorithm of [LT84] is applied on P. The transformation proceeds by rewriting P using one of the following rewrite rules, as long as P con-

tains formulas which are not normal clauses. The procedure returns $\mathcal{T}_d \cup P \cup \{\neg false\}$.

- *A formula* H :- $\neg F$ *or* H :- $L_1 \wedge \ldots \wedge \neg F \wedge \ldots \wedge L_n$ *in which* F *is not an atom, is simplified by moving the negation inside* F. *I.e.,* F *is replaced by a formula* F' *which is obtained from* F *in the normal way:*

  $$\begin{array}{ll} \textit{if } F = G \vee H & \textit{then } F' = \neg G \wedge \neg H \\ \textit{if } F = G \wedge H & \textit{then } F' = \neg G \vee \neg H \\ \textit{if } F = \exists X : G & \textit{then } F' = \forall X : \neg G \\ \textit{if } F = \forall X : G & \textit{then } F' = \exists X : \neg G \\ \textit{if } F = \neg\neg G & \textit{then } F' = G \end{array}$$

- *A formula of the form* H :- $F_1 \wedge \ldots \wedge \exists X : F_i \wedge \ldots \wedge F_n$ *is simplified to*

  $$H \text{ :- } F_1 \wedge \ldots \wedge F_i \wedge \ldots \wedge F_n$$

- *A formula of the form* H :- $F_1 \wedge \ldots \wedge (F \vee G) \wedge \ldots \wedge F_n$ *is replaced by the rules*

  $$H \text{ :- } F_1 \wedge \ldots \wedge F \wedge \ldots \wedge F_n$$
  $$H \text{ :- } F_1 \wedge \ldots \wedge G \wedge \ldots \wedge F_n$$

- *A formula of the form* H :- $F_1 \wedge \ldots \wedge \forall X : F_i \wedge \ldots \wedge F_n$ *is replaced by the rules:*

  $$H \text{ :- } F_1 \wedge \ldots \wedge \neg p(X_1, \ldots, X_n) \wedge \ldots \wedge F_n$$
  $$p(X_1, \ldots, X_n) \text{ :- } \exists X : \neg F_i$$

  Here p/n *is a new predicate and* $X_1, .., X_n$ *are the free variables of* $\forall X : F_i$.

[Dec89] has extended the transformation of [LT84] by a pre-processing step, which can often avoid floundering. Another trivial case where better results may be obtained is when $\mathcal{T}$ contains expressions which are completed definitions. Obviously, it is better to add the definition directly. For example, in the FOL situation calculus used in [Rei92], equivalences occur which are completed definitions of predicates in the LP version of situation calculus.

[LT84] proves that the algorithm always terminates. The transformed program contains precisely the same undefined predicates as the source theory. The produced program P on itself is hierarchical. Note however that the body of the clauses in P may contain literals of the defined predicates of $\mathcal{T}_d$. Hence, P may depend on $\mathcal{T}_d$ which is in general not hierarchical. For the following theory:

$$\begin{array}{l} \mathcal{T}_1 = \{\ (p \vee q) \wedge (\neg p \vee \neg q) \\ \qquad r \text{ :- } p \\ \qquad r \text{ :- } q\ \} \end{array}$$

the procedure produces :

$$\mathcal{T}'_1 = \{\ \neg false$$
$$false \text{ :- } \neg p, \neg q$$
$$false \text{ :- } p, q$$
$$r \text{ :- } p$$
$$r \text{ :- } q\ \}$$

Theorem 4.9.1 *Given is a theory $\mathcal{T} = \mathcal{T}_d \cup \mathcal{T}_c$ based on $\mathcal{L}$. Assume that algorithm 4.9.1, transforms $\mathcal{T}$ to $\mathcal{T}' = \mathcal{T}_d \cup P \cup \{\neg false\}$ and $\mathcal{T}'$ is based on $\mathcal{L}'$. Then $<\mathcal{L}', \mathcal{T}'>$ is an* elementary extension *of $<\mathcal{L}, \mathcal{T}>$ wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$.*

Proof The transformation can be seen as an iteration producing a sequence $P_1, \ldots, P_m$ of sets of formulas where $P_1 = \{false \text{ :- } \neg F_1, \ldots, false \text{ :- } \neg F_k\}$ and P_m is the resulting set of normal clauses. Consider the following sequence of theories:

$$\mathcal{T}_0 = \mathcal{T}$$
$$\mathcal{T}_i = \mathcal{T}_d \cup comp_3(P_i) \cup \{\neg false\} (0 < i \leq m)$$
$$\mathcal{T}_{m+1} = \mathcal{T}' = \mathcal{T}_d \cup P_m \cup \{\neg false\}$$

Each of these theories is based on a language $\mathcal{L}_i$ where $\mathcal{L}_0 = \mathcal{L}$ and $\mathcal{L}_m = \mathcal{L}_{m+1} = \mathcal{L}'$. We prove that for each $i \geq 0$, $\mathcal{T}_{i+1}$ is an elementary extension of $\mathcal{T}_i$ wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$. Clearly, this implies that $\mathcal{T}'$ is an elementary extension of $\mathcal{T}$ wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$.

First take $i = 0$. Take any model of $\mathcal{T}_0$. It can be extended to $\mathcal{L}_1$ by assigning $\mathcal{H}_M(false) = \mathbf{f}$. This is a model of $\mathcal{T}_1$. Vice versa, any model of $\mathcal{T}_1$ obviously is a model of $\mathcal{T}_0$ (wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$).

Let $0 < i \leq m$. It can easily be verified that if any except the last transformation step (for universal quantifiers) of algorithm 4.9.1 is applied, then $comp_3(P_i)$ and $comp_3(P_{i+1})$ are based on the same language and are logically equivalent in the sense that they have precisely the same models. Therefore, any model of $\mathcal{T}_i$ (wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$) is a model of $\mathcal{T}_{i+1}$ and vice versa.

So assume that the last transformation step is applied. A formula of the form

$$H \text{ :- } F_1 \wedge \ldots \wedge \forall X : F_j \wedge \ldots \wedge F_k$$

is replaced by the rules

$$H \text{ :- } F_1 \wedge \ldots \wedge \neg p(X_1, \ldots, X_n) \wedge \ldots \wedge F_n$$

$$p(X_1, \ldots, X_n) \text{ :- } \exists X : \neg F_j$$

where p/n is a new relation symbol. Consider any model M' of $\mathcal{T}_{i+1}$ (wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$) and any simple fact $p(x_1, \ldots, x_n)$. In particular M' is

a model of $<\mathcal{L}_{i+1}, comp_3(P_{i+1})>$. Let V be the variable assignment $\{X_1/x_1, \ldots, X_n/x_n\}$. We have that $\mathcal{H}_{M'}(\neg p(x_1, \ldots, x_n)) = \mathcal{H}_{M'}(V(\forall X : F_j)$. This obviously implies that the restriction of M' to $\mathcal{L}_i$ is a model of $comp_3(P_i)$, and hence a model of $\mathcal{T}_i$ (wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$).

Vice versa, let M be a model of $\mathcal{T}_i$ (wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$) and hence of $<\mathcal{L}_i, comp_3(P_i)>$. Note that for each ground instance of

$$p(X_1, \ldots, X_n) \text{ :- } \exists X : \neg F_j$$

the right side is based on $\mathcal{L}_i$ and therefore, its truth value is completely determined by M. We extend M to an interpretation M' of $\mathcal{L}_{i+1}$ as follows: for each simple fact $p(x_1, \ldots, x_n)$ with associated V:

$$\mathcal{H}_{M'}(p(x_1, \ldots, x_m) = \mathcal{H}_M(V(\exists X : \neg F_j)$$

It is trivial to see that M' is a model of $comp_3(P_{i+1})$ and a model of $\mathcal{T}_{i+1}$ (wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$).

Finally, take $i = m$. We must show here that an interpretation M of $\mathcal{L}'$ is a model of $\mathcal{T}_m = \mathcal{T}_d \cup comp_3(P_m) \cup \{\neg false\}$ (wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$) iff M is a model of $\mathcal{T}_{m+1} = \mathcal{T}_d \cup P_m \cup \{\neg false\}$ (wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$).

First assume that M is a model of $\mathcal{T}_{m+1}$ wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$. M is a model of $\neg false$. Note that the predicates which are defined in P_m do not appear in $\mathcal{T}_d$. As a consequence, M is a model of $\mathcal{T}_d$ wrt $(\mathcal{DJS})(\mathcal{PJS})$ $(\mathcal{JS})$. It remains to show that M is a model of $comp_3(P_m) \cup \{\neg false\}$. By theorem 4.3.2, M is a directly justified model of $\mathcal{T}_d \cup P_m$, and by theorem 4.4.1(b) of $comp_3(\mathcal{T}_d \cup P_m)$. Since $comp_3(P_m)$ is comprised in $comp_3(\mathcal{T}_d \cup P_m)$, M is a model of $comp_3(P_m)$. Together, we find that M is a model of $\mathcal{T}_m$ wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$.

Second, assume that M is a model of $\mathcal{T}_m$ wrt $(\mathcal{DJS})(\mathcal{PJS})(\mathcal{JS})$. When M is a directly justified model of $\mathcal{T}_m$, then it follows easily from theorem 4.4.1 that M is a directly justified model of $\mathcal{T}_{m+1}$.

So assume that M is a model of $\mathcal{T}_m$ wrt $(\mathcal{PJS})$ $(\mathcal{JS})$. We must prove that for each simple fact F of $\mathcal{L}' \setminus \mathcal{L}$, it holds that

$$\mathcal{H}_M(F) = (SV_{PJ}(M, F))(SV_J(M, F))$$

We prove the theorem for $\mathcal{PJS}$. The proof for $\mathcal{JS}$ is totally analogous.

M is a directly justified model of $\mathcal{T}_m$ and hence, M is a directly justified model of $\mathcal{T}_{m+1}$. So, $\mathcal{H}_M(F) = SV_{DJ}(M, F)$. It suffices to prove that $SV_{DJ}(M, F) = SV_{PJ}(M, F)$. Note that P_m is a hierarchical program. Unfortunately, $\mathcal{T}_d \cup P_m$ is not hierarchical, so that we cannot use proposition 4.3.3 to prove this equality. The proof is by induction of the stratum of F.

We define the stratum of p/n as 0 when p/n belongs to $\mathcal{L}$. For a predicate p/n of $\mathcal{L}' \setminus \mathcal{L}$, we define the stratum of p/n as the maximum of the strata of predicates occurring in the body of the clauses of the definition of p/n incremented with one.

Let i be the stratum of F. Nothing is to prove when $i = 0$. Assume that it has been proven for facts F' with stratum $< i$ that $\mathcal{H}_M(F') = SV_{PJ}(M, F')$.

First we prove that $\mathcal{H}_M(F) \leq SV_{PJ}(M, F)$. Let J be a direct justification of F with maximal value. We have that $\mathcal{H}_M(F) = SV_{DJ}(M, F) = val_M(J) = min\{\mathcal{H}_M(F') \mid F' \in J\}$. Note that each $F' \in J$ is of a lower stratum and hence, $\mathcal{H}_M(F') = SV_{PJ}(M, F')$. So for each defined fact $F' \in J$, there exists a partial justification $J_{F'}$ such that $\mathcal{H}_M(F') = val_M(J_{F'})$. We use this to construct a partial justification J' of F. If F is a positive (negative) fact, then J' is obtained by concatenating J with $J_{F'}$ for each positive (negative) fact $F' \in J$. It is easy to see that $val_M(J') = val_M(J)$. Hence $\mathcal{H}_M(F) = val_M(J') \leq SV_{PJ}(M, F)$.

This proves that for each fact F of the i'th stratum: $\mathcal{H}_I(F) \leq SV_{PJ}(I, F)$. Assume that for some fact F, $\mathcal{H}_I(F) < SV_{PJ}(I, F)$. $\sim F$ belongs to the same stratum. By theorem 4.3.1, $\mathcal{H}_I(\sim F) = \mathcal{H}_I(F)^{-1} > SV_{PJ}(I, F)^{-1} = SV_{PJ}(I, \sim F)$. This is a contradiction.

□

A direct but important consequence of the theorem is that any FOL theory $\mathcal{T}$ can be transformed into an incomplete logic program and FOL axiom $\neg false$. This is because the models of $\mathcal{T}$ wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$ are precisely the models of $\mathcal{T}$ wrt 2-valued classical model theory.

Proposition 4.9.1 *Let $\mathcal{T}$ be a first order theory, $\mathcal{T}$' its translation by algorithm 4.9.1. Then $\mathcal{T}'$ wrt to $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$ is an elementary extension of $\mathcal{T}$ wrt the FOL model semantics.*

The theorem asserts that any classical model of $\mathcal{T}$ can be extended to a model of $\mathcal{T}$' wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$ and vice versa the restriction of any model of $\mathcal{T}$' wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$ to the symbols of the language of $\mathcal{T}$ is a classical model of $\mathcal{T}$. This implies a very strong form of equivalence, for example that for any formula F based on the original language: F is implied by $\mathcal{T}$ wrt the FOL model semantics iff F is implied by $\mathcal{T}$' wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$.

The theorem effectively integrates First Order Logic and Logic Programming. For FOL, the gain one might expect from this integration, is on the computational level: the efforts in logic programming on implementation and transformation may render feasible implementations possible. For logic programming, the gain is on the representational level: existing work in FOL can be transformed to LP.

Theorem 4.9.1 has some other important implications. One application is that an abductive procedure, developed for pure incomplete logic programs without FOL axioms, can be used to deal with theories with FOL axioms. It suffices to apply the transformation on the set of FOL axioms and to add the FOL axiom $\neg false$ to the query.

Second and related, an abductive procedure can be used as a model generator for classical logic (with $FEQ(\mathcal{L})$). For any theory $\mathcal{T}$, an abductive procedure can execute the goal $\leftarrow \neg false$ on the transformation of $\mathcal{T}$. If it fails, it has proven the inconsistency of $\mathcal{T}$. If it succeeds, it has found a finite model of $\mathcal{T}$. Note that the transformation of $\mathcal{T}$ is a hierarchical program, for which $\mathcal{DJS}$, $\mathcal{PJS}$ and $\mathcal{JS}$ coincide.

Theorem 4.9.2 *Let $\mathcal{T}$ be a FOL theory based on $\mathcal{L}$, its transformation being $\mathcal{T}_d + \{\neg false\}$, based on a language $\mathcal{L}'$ extending $\mathcal{L}$. Let Δ be an abductive solution based on $\mathcal{L}$ for the query $\leftarrow \neg false$ (wrt $\mathcal{DJS}$).*

It holds that Δ is a (FOL) model of $\mathcal{T}$ + FEQ + SDCA.

Proof Δ defines a two-valued incomplete Herbrand interpretation I_Δ for all undefined predicates of $\mathcal{T}_d$. I_Δ can be extended to a directly justified model M of $\mathcal{T}_d$. Obviously, M is a directly justified model of $\mathcal{T}_d + \Delta$. Hence $M \models \neg false$. Therefore, for any formula F of $\mathcal{T}$, $M \models F$. Hence, I_Δ which is the restriction of M to $\mathcal{L}$, is a model of $\mathcal{T}$. □

This theorem shows again the intimate relationship between abduction and model generation. Whereas in chapter 3, we showed that abduction can be simulated by a model generation procedure, here we find the reverse theorem.

The theorem shows also how hard the problem of abduction is.

Proposition 4.9.2 *Even for a hierarchical program, the problem of finding a (possibly infinite) abductive solution is undecidable.*

Proof The proposition follows directly from theorem 4.9.2 and the fact that it is undecidable whether a FOL theory has a model. □

Finally, we mention other interesting approaches for representing incomplete knowledge in LP. In disjunctive logic programming, the head of rules can contain disjunctions of atoms [MR90]. A more recent approach is extended logic programming [GL90a]. This formalism was used for representing incomplete knowledge in [GL92]. At present, a good comparison between all alternative formalisms and semantics is lacking and is a subject for future research. However, a point that can be made here is that the justification semantics is the only semantics based on general interpretations. Therefore, it is the only semantics which allows to represent incomplete knowledge on the problem domain.

4.10 Expressivity of logic programs

The LP-FOL-formalism under justification semantics not only integrates FOL logic, it provides meaningful extra expressivity. A convincing example is natural number arithmetic. Consider the language $\mathcal{L}_{nat}$ with functors $0/0$ and $s/1$ and predicates $plus/3$ and $times/3$. Consider the following definite program.

$$\begin{aligned} P_{nat} = \{\ & plus(0, X, X) \text{ :-} \blacksquare \\ & plus(s(X), Y, s(Z)) \text{ :-} plus(X, Y, Z) \\ & times(0, X, 0) \text{ :-} \blacksquare \\ & times(s(X), Y, Z) \text{ :-} times(X, Y, Z1), plus(Y, Z1, Z) \quad \} \end{aligned}$$

This is a definite logic program and has a unique least Herbrand model, which is obviously $\mathbb{N}, +, \times$. Under least Herbrand model semantics, P_{nat} has only this model, and therefore it is a *complete axiomatisation* of $\mathbb{N}, +, \times$. By adding $SDCA + FEQ$ to P_{nat}, $\mathbb{N}, +, \times$ is the unique justified model (modulo isomorphism) and we obtain a complete axiomatisation under justification semantics too. By Gödel's incompleteness theorem [K. 31], natural number arithmetic has no complete axiomatisation in first order logic.

This example indicates also that a high price is to be paid for this extra expressivity. By Church's theorem [Chu36], we know that $\mathbb{N}, +, \times$ is undecidable, hence no complete proof procedure for LP exists. This problem occurs in most LP semantics, since for definite programs, with exception of the completion semantics, most semantics coincide with the least Herbrand model semantics. The lack of a complete proof procedure was previously reported in [Prz89]. It is often argued that a formal language for automated problem solving and computing should have a complete and efficient proof procedure. The undecidability of deduction for the LP formalism raises the fundamental question whether the LP formalism is a suitable language for computing and automated problem solving.

There are at least two reasons why the undecidability in the LP formalism does not prevent the language to be of practical interest. First of all, *undecidability* is a worst case complexity. The undecidability is a problem of only a subclass of theories in the space of problems which can be formulated in the logic. This subclass may be disjunct with the class of problems which arise in practice. One example of a theoretically useless system is the simplex algorithm for linear programming. Despite its exponential complexity (which is almost as bad as undecidability), the algorithm proves to be very useful. There exist polynomial algorithms for linear programming, but these are not only more complex but for most applications also less efficient than the simplex algorithm.

Secondly, a more fundamental reason is that the complexity of a computational problem is inherent rather to the problem than to the logic in which it is described. Therefore, allowing only less expressive but efficient logics has something of an ostrich policy. In principle, simple problems may be executed in expressive logics as efficiently as in less expressive logics. An example where this seems to happen

is constraint logic programming. In constraint logic programming, a general purpose theorem prover for logic programming is extended with a number of highly specialised problem solvers which can only solve specific subclasses of problems. These specific subclasses are "simple" in the sense that efficient procedures exist to solve them or that a decision procedure exist. This way CLP combines high expressivity and remarkable efficiency [Van89].

For complex problem domains (take the natural numbers), an expressive logic has the advantage of allowing an accurate representation. Hard problems cannot be solved by the theorem prover but probably a large class of simpler problems can be solved. On the other hand, such problem domains cannot even be represented correctly in efficient logics, which forces the knowledge engineer to find approximations of his domain. The hard problems cannot be solved because the approximation is to imprecise.

It is interesting to investigate how the domain of the natural numbers is approximated in the classical FOL theory on natural number arithmetic, the so called Peano axioms. We recall the formulation from [Men72]. It uses operators $+$ and $\times$ instead of predicates $plus/1$ and $times/3$.

(S1) $X = Z \leftarrow X = Y \wedge Y = Z$
(S2) $s(X) = s(Y) \leftarrow X = Y$
(S3) $\neg 0 = s(X)$
(S4) $X = Y \leftarrow s(X) = s(Y)$
(S5) $X + 0 = X$
(S6) $X + s(Y) = s(X + Y)$
(S7) $X \times 0 = 0$
(S8) $X \times s(Y) = (X \times Y) + X$
(S9) $P[0] \wedge (\forall X : P[X] \rightarrow P[s(X)]) \rightarrow \forall X : P[X]$
where P[X] denotes any formula with free variable X

One easily verifies that axioms (S1)-(S4) are included in FEQ. Axioms (S5)-(S8) correspond to the clauses of P_{nat}. The most remarkable fact is the complete lack of induction axioms (S9) in $P_{nat} + SDCA + FEQ$, while on the other hand, there seems no formula in the Peano axioms which corresponds to the $SDCA$. Despite their rather different form, the induction axioms and the $SDCA$ are very related: $SDCA$ subsumes the induction axioms. To see this, assume that in some model I of the $SDCA$, for some open formula $P[X]$, $\forall X : P[X]$ does not hold. We should prove that the premise of the corresponding induction axiom is not satisfied either. We know that for each domain element x of I, there exists a natural number n such that $\tilde{I}(s^n(0)) = x$. Let n be the minimal number such that $P[s^n(0)]$ is not satisfied in I. Either $n = 0$ or $n > 0$ and $P[s^{n-1}(0)]$ is true. In both cases, the premise of the induction axiom is not satisfied.

This observation shows that the induction axioms must really be interpreted as approximations of the $SDCA$. This raises an observation on the computational

level. Since *SDCA* subsumes the induction axioms, each theorem which can be proven by mathematical induction is implied by the same theory with *SDCA* substituted for the induction axioms. This implies that under Herbrand model based semantics or equivalently for theories including *SDCA*+*FEQ*, theorem proving should incorporate a generalised form of mathematical induction for arbitrary languages.

We conclude this section with some observations of a more speculative nature. An intriguing question is what is the cause of LP's extra expressivity compared to FOL. Why can the DCA be expressed in logic programming (under justification semantics) but not in FOL? Below, we argue that FOL's inadequacy is due to its failure to express *inductively defined concepts*. The principle of *inductive definition* is an elementary way of defining concepts in mathematics. A remarkable observation, recognized in the mathematical logic community, but often unnoticed in the logic programming community, is that first order logic in general does not allow to represent an inductively defined concept in a precise way. As an example, consider the concept of a natural number. It can be inductively defined as follows:

> 0 is a natural number
> If n is a natural number, then $s(n)$ is a natural number.

This is a precise mathematical definition, yielding $\{0, s(0), s(s(0)), \ldots\}$. In FOL, it is impossible to construct a theory describing a predicate $nat/1$ which represents exactly the natural numbers. Of course, a theory $\mathcal{T}$ with clauses:

$$nat(0) \leftarrow$$
$$nat(s(X)) \leftarrow nat(X)$$

implies that for each term $s^n(0)$, $\mathcal{T} \models s^n(0)$. It is even easy to prove that this theory *defines* the natural numbers $nat/1$ in the following sense: $\mathcal{T} \models nat(x)$ iff $x \in \mathbb{N}$. However, for even easy queries about the natural numbers this does not suffice. For example, extend $\mathcal{T}$ with the following clauses for $odd/1$ and $even/1$:

$$even(0) \leftarrow$$
$$even(s(s(X))) \leftarrow even(X)$$
$$odd(s(0)) \leftarrow$$
$$odd(s(s(X))) \leftarrow odd(X)$$

In a correct representation of $nat/1$, it should be possible to prove that each natural number is either even or odd. $\mathcal{T}$ does not imply this. Consider the interpretation M with domain:

$$D = \{s^n(0), s^n(a) \mid n \in \mathbb{N}\}$$

and $\mathcal{H}_M$ defined as follows:

$$\{nat(s^n(0))^{\mathbf{t}}, nat(s^n(a))^{\mathbf{t}}, even(s^{2\times n}(0))^{\mathbf{t}}, odd(s^{2\times n+1}(0))^{\mathbf{t}} \mid n \in \mathbb{N}\}$$

M is a model of $\mathcal{T}$ but the formula $\forall X : nat(X) \rightarrow odd(X) \vee even(X)$ is not true. Notice that the problem is caused by additional undesired elements $s^n(a)$ in the interpretation of $nat/1$.

It does not suffice to take the completion $comp(\mathcal{T})$. $comp(\mathcal{T})$ comprises the completed definition of $nat/1$:

$$\forall X : nat(X) \leftrightarrow X = 0 \vee \exists Y : X = s(Y) \wedge nat(Y)$$

Consider the interpretation M with domain the disjunct union of the natural numbers $\mathbb{N}$ and the integer numbers $\mathbb{Z}$. $M(s/1)$ is defined as the union of the successor function on the natural numbers and the integer numbers. $\mathcal{H}_M$ is defined as follows:

$$\{nat(n)^{\mathbf{t}}, nat(z)^{\mathbf{t}}, even(2 \times n)^{\mathbf{t}}, odd(2 \times n + 1)^{\mathbf{t}} \mid n \in \mathbb{N} \wedge z \in \mathbb{Z}\}$$

M is a model of $comp(\mathcal{T})$, and $\forall X : nat(X) \rightarrow odd(X) \vee even(X)\}$ is still not true. Again, the problem is caused by the undesired elements in the interpretation of $nat/1$.

The problem with the two theories above is that they allow models in which more domain elements belong to $nat/1$ than the natural numbers. There is simply no way to avoid this in FOL. This fact is generally known in mathematical logic, but an explicit proof of the theorem precisely as we need it here, is hard to find, therefore, we give a proof in the following corollary.

Corollary 4.10.1 *There exists no consistent FOL theory $\mathcal{T}_{nat}$ describing a predicate $nat/1$ such that in each model M, $M \models nat(x) \Leftrightarrow \exists n \in \mathbb{N} : \tilde{M}(s^n(0)) = x$ and if $n, m \in \mathbb{N}$ and $n \neq m$ then $\tilde{M}(s^n(0)) \neq \tilde{M}(s^m(0))$.*

Proof Assume that a FOL theory $\mathcal{T}_{nat}$ exists, based on a language $\mathcal{L}$, which satisfies the conditions in the theorem. We prove that we can extend $\mathcal{T}_{nat}$ to a theory which allows to decide all properties of $\mathbb{N}, +, \times$. This contradicts Church's theorem.

Below we define $\mathcal{L}_{nat}$ as the language with functors 0, $s/1$ and predicate symbols $plus/3$ and $times/3$. The language $\mathcal{L}$ of $\mathcal{T}_{nat}$ comprises obviously 0, $s/1$ and $nat/1$. We assume without loss of generality that $\mathcal{L}$ does not contain the predicate symbols $plus/3$ and $times/3$. Below we denote the set of terms $\{0, s(0), s(s(0)), \ldots\}$ by $\mathbb{N}$. Observe that for any model M of $<\mathcal{L}, \mathcal{T}_{nat}>$, the restriction of $\tilde{M}$ to $\mathbb{N}$ defines a mapping h of $\mathbb{N}$ into the domain of M such that for any two terms n and m, if $n \neq m$ then $h(n) \neq h(m)$. Consider an extension $\mathcal{L}$' of $\mathcal{L}$ with the predicates $plus/3$ and $times/3$. Clearly, any model of $<\mathcal{L}, \mathcal{T}_{nat}>$ extended in arbitrary way to $\mathcal{L}$' is a model of $<\mathcal{L}', \mathcal{T}_{nat}>$. Hence, $<\mathcal{L}', \mathcal{T}_{nat}>$ is consistent.

Now observe that any model M of $<\mathcal{L}', \mathcal{T}_{nat}>$ defines a Herbrand interpretation M_h of $\mathcal{L}_{nat}$, in the following way: for any $plus/3$ or $times/3$

atom A, $\mathcal{H}_{M_h}(A) = \mathcal{H}_M(A)$. We call M_h the associated Herbrand interpretation of M. Vice versa, given some Herbrand interpretation M_h of $\mathcal{L}_{nat}$ and any model M of $<\mathcal{L}, \mathcal{T}_{nat}>$, M can be extended to an interpretation M' of $\mathcal{L}'$ in a way such that for each atom A of $HB(\mathcal{L}_{nat})$, it holds that $\mathcal{H}_{M_h}(A) = \mathcal{H}_{M'}(A)$. The extension is constructed as follows: for each simple fact A of $plus/3$ or $times/3$, if there exists $B \in HB(\mathcal{L}_{nat})$ such that $\tilde{M}(B) = A$, define $\mathcal{H}_{M'}(A) = \mathcal{H}_{M_h}(B)$; if such a B does not exist define $\mathcal{H}_{M'}(A) = \mathbf{f}$. We call M' the extension of M by M_h.

So assume that M is a model of $<\mathcal{L}', \mathcal{T}_{nat}>$ with associated Herbrand interpretation M_h. For any formula F based on $\mathcal{L}_{nat}$, there is a related formula F^r, called the range restricted form of F, based on $\mathcal{L}$', such that $\mathcal{H}_{M_h}(F) = \mathcal{H}_M(F^r)$. F^r can be constructed by substituting each subformula $\forall X : F_c$ in F for $\forall X : nat(X) \rightarrow F_c$ and substituting $\exists X : F_c$ for $\exists X : nat(X) \wedge F_c$. That M_h and M assign equal truth values to F and F^r can easily be proved by induction on the structure of F.

Now consider again the definite program P_{nat}, defined earlier in this section. It is easy to see that P_{nat} is an acyclic program: a level mapping is given by defining $\|plus(n, m, z)\| = n$ and $\|times(n, m, z)\| = n + m$. For acyclic definite programs, [AB90] proves that the least Herbrand model is the only Herbrand model of the completion $comp(P)$. Consider the theory $\mathcal{T}_{+\times}$ consisting of $comp(P_{nat}) \setminus FEQ(\mathcal{L}_{nat})$, and the theory $\mathcal{T}^r_{+\times}$ consisting of the range restricted formulas of $\mathcal{T}_{+\times}$. Consider also the theory $\mathcal{T}_{\mathbb{N},+,\times} = \mathcal{T}_{nat} + \mathcal{T}^r_{+\times}$. First of all, note that this theory is consistent. Indeed, the extension of a model of $\mathcal{T}_{nat}$ by $\mathbb{N}, +, \times$ satisfies $\mathcal{T}_{nat}$ and satisfies $\mathcal{T}^r_{+\times}$ because $\mathbb{N}, +, \times$ satisfies $\mathcal{T}_{+\times}$. Moreover, for each model M of $\mathcal{T}_{\mathbb{N},+,\times}$, its associated Herbrand model M_h satisfies $\mathcal{T}_{+\times}$, and hence M_h is $\mathbb{N}, +, \times$. We find that for every proposition F based on $\mathcal{L}_{nat}$, $\mathbb{N}, +, \times \models F$ iff $\mathcal{T}_{\mathbb{N},+,\times} \models F^r$. This implies that $\mathcal{T}_{\mathbb{N},+,\times}$ allows to decide every proposition F on $\mathbb{N}, +, \times$: run a complete FOL theorem at the same time on F^r and $\neg F^r$ by coroutining, until one of both formulas is proven. Since $\mathbb{N}, +, \times \models F$ or $\mathbb{N}, +, \times \models \neg F$ and since FOL is semi-decidable, eventually the theorem prover will stop with a proof for either F^r or $\neg F^r$. This is in contradiction with Church's undecidability theorem [Chu36][8].

□

The merit of this corollary is that it shows very precisely what goes wrong with FOL to represent the natural numbers. The fact that even simple inductively defined concepts cannot be expressed correctly in FOL, implies that many common

[8] The proof of this theorem holds only when $\mathcal{T}_{nat}$ is a recursive theory: i.e. when the axioms of $\mathcal{T}_{nat}$ can be enumerated by an algorithm. Only for such theories, a complete FOL theorem prover can be build. However, the theorem holds even in the more general case of an arbitrary non-recursive theory. The proof must then be given based on Ultrafilters [Den92]

concepts in mathematics and in informatics cannot be correctly represented in FOL. Is the LP formalism an accurate language to describe inductive definitions? One could expect this since this intuitive notion of *constructive definition* seems very related to this concept of *inductive definition*. Of course the answer to the question cannot be given without a precise definition of what is an inductive definition. The notion of inductive definition is the subject of an area in mathematics [Acz77]. In [Acz77], an inductive definition Φ is defined as a set of rules of the form:

$$X \rightarrow x$$

where X is a (possibly infinite) set of premises and x is the conclusion. With an inductive definition Φ, an operator can be associated, which maps a set A into the set of conclusions x for which there exist a rule $X \rightarrow x$ in Φ with premises in A. A set X is closed under Φ iff $\Phi(A) \subseteq A$. The inductively defined set by Φ is the least closed set of Φ. [Fef70] has proposed an extension of FOL by inductive definitions, using a set of second order axioms. The resulting formal system can be shown to be stronger than first order logic but weaker than second order logic.

There is an obvious correspondence between an inductive definition and a definite program, between rules of an inductive definition and ground instances of the clauses of a definite program, between the inductive definition as an operator and the T_P operator associated to a definite program, and between the inductively defined set by an inductive definition and the least Herbrand model of the definite program. Hence, a definite program can be interpreted as an inductive definition in the mathematical sense. The normal and general logic program formalism is both extending and extended by these mathematical inductive definitions: LP is extended since in general, the rules in mathematical inductive definitions are allowed to contain an infinite number of premises; LP extends the mathematical inductive definitions because LP allows negative premises to occur in the rules. A deep investigation of the concept of inductive definition in the presence of negative premises might ultimately result in a precise definition of what is a constructive definition and may lead to a better understanding and better motivation of the semantics of logic programs. This subject falls beyond the scope of this thesis and is a subject for future research.

4.11 Summary

We summarize the content of the chapter. On the technical level, we have investigated the different notions of justification found in completion, stable and well-founded semantics. The framework is for complete and incomplete logic programs, and is based on general 3-valued interpretations. This work considerably extends the work of [Fag90], which investigates only the stable semantics for complete logic programs, using 2-valued Herbrand interpretations. Further on, we have

shown that any set of FOL axioms can be transformed to an incomplete logic program. This implies that a simple abductive procedure developed for incomplete programs without FOL axioms needs no extension to deal with FOL axioms, and that such a procedure can be used as a model generator for first order logic.

More on the conceptual level, the framework shows that only in the well-founded semantics and its extension, the justification semantics, a true positive fact does not depend on itself. We therefore have proposed to view logic programs as sets of *constructive definitions*. This imposes the view on **u** as *locally inconsistent*. The result is a logic which is *essentially two-valued*: theories with 3-valued models are considered as containing bugs and a predicate and its negation are still seen as *complementary*. We have argued how incomplete knowledge, both on the predicates and on the problem domain entities, can be represented in incomplete logic programs under justification semantics. Last but not least, this work integrates first order logic within logic programming.

Chapter 5

An abductive procedure for normal incomplete programs

5.1 Introduction

Negation as failure and abduction have been recognized as important forms of non-monotonic reasoning [Kow90], [CM85], [Poo88]. They have been shown useful for fault diagnosis [CM85], natural language understanding [CM85], knowledge assimilation [KM90a] and default reasoning [EK89], [Poo88]. Here we present a technical contribution to the area. A general procedure for logic programs is proposed which integrates both negation as failure and abduction. This procedure resulted from an attempt to integrate the techniques of chapter 3 in a useful abductive procedure for temporal reasoning. Temporal reasoning is an excellent domain for testing non-monotonic reasoning techniques because of the *frame problem*: "the problem of representing the tendency of facts to endure over time" [HM87]. Hanks and McDermott used the famous Yale turkey shooting problem (YTS) to show that well-known non-monotonic reasoning systems as McCarthy's circumscription [McC80], Reiter's default logic [Rei80] and McDermott's non-monotonic logic [McD82] failed to represent the frame axiom correctly.

Negation as failure was not considered in this study. Nevertheless, the frame axiom has a correct representation in situation calculus or event calculus with negation as failure. It has been shown that the YTS representation in these formalisms solves the problem correctly [AB90], [EK89], [Eva89].

Negation as failure alone is not sufficient for representing many temporal reasoning problems. A major restriction is its inability of representing *incomplete*

knowledge. The original event calculus only supports the prediction of a goal state, starting from a complete description of the initial state and the set of events. In many problems, either the initial state or the sequence of events are unknown. In planning, for example, the set of events is the subject of the search, and thus, a priori unknown. A solution to this problem is to apply incomplete event calculus and abduction [Esh88], [Sha89]. In planning problems, the predicates which describe the events, i.e. $happens/1, act/2$ and $<$ are undefined. An abductive solution for a problem, given the goal state, consists of a set of events and their order.

A sound abductive procedure for normal programs has been defined in [KM90a]. This procedure is based on the view that negation by failure is a special form of abduction, a view first presented in [Esh88] (see also section 4.7). Unfortunately, it turns out that for the purpose of temporal reasoning and planning, this procedure is not powerful enough. The limitation is that an abducible literal in a goal can only be abduced when it is ground. To see the problem, consider (a simplification of) a typical event calculus clause:

$$p \leftarrow happens(E), act(E, initiate\_p)$$

Observe that the variable E occurs only in abducible atoms. Therefore, when executing the goal $\leftarrow p$, the atoms $happens(E)$ and $act(E, initiate\_p)$ will never become ground and can never be selected for abduction. Thus, the procedure *flounders* on these non-ground abductive goals: it terminates without a complete computation.

In the past, special abductive procedures which do not suffer from this limitation, have been presented for temporal reasoning with abductive event calculus [Esh88], [Sha89]. Recently, [Mis91b, Mis91a] described an implementation of such a planner based on a special purpose abductive procedure. However, the procedure in [Esh88] is for definite programs with integrity constraints; no formalisation is given and soundness and completeness results are lacking. These results are also lacking for the procedure in [Sha89]. In [Mis91b, Mis91a], the abductive procedure is formalised and its correctness is proven for a specific class of planning problems, but the procedure is unsound in the general case. Moreover, as the authors argue, their treatment of non-ground abductive goals for which finite failure must be proven, is very inefficient.

We present an improved treatment of non-ground abductive goals which has been incorporated in a procedure, called SLDNFA. We prove its sound- and completeness. Although the inspiration for the design of SLDNFA stems from temporal reasoning, we formulate it in full generality and it can be applied in any domain where abduction is useful.

In section 5.2, we present the intuitions behind SLDNFA and define the basic inference operators. In section 5.3 we formalise SLDNFA and indicate its relation to SLDNF. In section 5.4, 5.5 and 5.6, we present the proofs of the soundness and completeness. In section 5.7, variants of SLDNFA are presented, which yield other

completeness results. Finally, we end with a discussion. A short paper with the main definitions and results of this chapter was published as [DD92b].

5.2 Basic computation steps in SLDNFA

The SLDNFA procedure is an abductive procedure for normal abductive programs. For logic programs without negation as failure, the SLD-procedure can be extended easily to an abductive procedure [CP86], [FG85] (see also chapter 3). Combining abduction with negation as failure is less straightforward. SLDNFA is an extension of the well-known SLDNF procedure [Llo87] for complete logic programs. The SLDNF procedure can be viewed as the process of proving an initial goal by constructing a set $\mathcal{PG}$ of goals that must succeed and a set $\mathcal{NG}$ of goals that must fail. SLDNF tries to reduce goals in $\mathcal{PG}$ to the empty goal $\Box$ and tries to build a finitely failed tree for the goals in $\mathcal{NG}$. When a ground negative literal $\neg A$ is selected in a goal in $\mathcal{PG}$, $\leftarrow A$ is added to $\mathcal{NG}$ and vice versa, when a ground negative literal $\neg A$ is selected in a goal in $\mathcal{NG}$, $\leftarrow A$ is added to $\mathcal{PG}$. In the sequel, we call a goal from $\mathcal{PG}$ a *positive goal* and a goal from $\mathcal{NG}$ a *negative goal*. Keep in mind that these names refer to the mode of execution for the goal, not to the sign of the literals in the goal. For SLDNFA, this computation scheme must be extended for the case that an abducible atom is selected in a positive or negative goal. Let us investigate the problems that may rise.

The case where an abducible atom A is selected in a positive goal, can be solved by skolemising the atom and adding the resulting atom to Δ. Skolemising A means that for each variable X appearing in A, a new constant sk is created which does not belong to the language $\mathcal{L}$ of the program and which is assigned to X. It is this solution for dealing with non-ground abducible atoms which has been applied in [Esh88], [Sha89] and [Mis91a].

The definitions below formalise this properly. Let $\mathcal{V}$ be the countably infinite set of variables of the first order language $\mathcal{L}$ and $\mathcal{SK}$ an infinite set of skolem constants not appearing in $\mathcal{L}$ ($\mathcal{SK} \cap \mathcal{L} = \phi$). Below we extend the notion of substitution by allowing skolem constants to appear in the domain of a substitution.

Definition 5.2.1 (skolemisation) *A skolemisation mapping $\mathcal{D}$ is a one-to-one function from $\mathcal{V}$ onto $\mathcal{SK}$. The deskolemisation mapping is its inverse $\mathcal{D}^{-1}$.*

The skolemising substitution *θ of a term or formula or set of terms or formulas with free variables $\overline{X}$ is the substitution $\{X_i/D(X_i) \mid X_i \in \overline{X}\}$.*

A deskolemising substitution *is a substitution consisting a finite set of pairs $D(X)/X$. The inverse of a skolemising substitution θ is a deskolemising substitution and is denoted θ^{-1}.*

Definition 5.2.2 (abduction) *Let Q be a goal $\leftarrow L_1, \ldots, L_m, \ldots, L_k$, with L_m an abducible atom. Q' is derived from Q by abducing L_m using the skolemising*

substitution θ iff θ is a skolemising substitution for L_m and Q' is the goal

$$\theta(\leftarrow L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_k)$$

As can easily be imagined, the introduction of skolem constants causes additional problems with the unification, both in positive and negative goals. An example illustrates the problem in positive goals. Consider the definite clause:

$$p(f(g(Z), V)) \text{ :-}$$

and the query $\leftarrow r(X), p(X)$ where $r/1$ is an abducible predicate. Consider the following partial derivation (selected literals are underlined):

$\mathcal{PG} = \{\leftarrow \underline{r(X)}, p(X)\}$ *Abduction*
$\mathcal{PG} = \{\leftarrow \underline{p(sk)}\}$, $\Delta = \{r(sk)\}$

Solving the positive goal $\leftarrow p(sk)$ is not trivial: classical unification cannot be applied since the unification of sk and $f(g(Z), V)$ would fail, which is not what is intended. The solution proposed by Eshghi [Esh88] was to introduce the equality predicate as an abducible predicate and to add the theory of FEQ as integrity constraints. When a skolem constant sk is to be unified with a term t, the equality fact $sk = t$ is skolemised and abduced explicitly and the consistency of $sk = t$ with other abduced facts and FEQ is checked. When for example there is a second abduced fact $sk = t'$, then the consistency of $t = t'$ wrt FEQ must be checked. Unfortunately, the explicit treatment of equality under FEQ produces a lot of overhead. Because of this problem, Shanahan [Sha89] and later Missiaen [Mis91b, Mis91a] gave up this solution and proposed a seemingly different solution, namely to extend unification for skolems. In a first phase, extended unification performs unification, treating skolem constants as variables; in a second phase, it skolemises the terms bound to the original skolem constants. In the example, the extended unification procedure substitutes sk by $f(g(sk_1), sk_2)$ and returns $\Delta = \{r(f(g(sk_1), sk_2))\}$.

Interestingly, the duality framework indicates a close relation between both approaches. Note that Eshghi's problem of keeping a set of ground equality facts consistent wrt FEQ also occurred in chapter 3 during model generation under FEQ. This suggests that the techniques developed in chapter 3 for dynamic completion and normalisation, can be applied to efficiently implement Eshghi's proposal. Remember also that the dynamic completion under FEQ is dual to unification and that the normalisation is dual to applying a substitution. As a consequence, the completion can easily be integrated in the unification procedure. The extended unification procedure produces a kind of substitution which is the union of a variable substitution (the mgu) and a ground complete term rewriting system. The normalisation of the terms can be integrated in the procedure of applying the variable substitution. This way the duality framework shows that the proposal of

[Sha89] can be seen as an efficient implementation of the procedure in [Esh88] and [Mis91b, Mis91a].

Below we extend unification and resolution in this spirit.

Definition 5.2.3 *An equality set based on $\mathcal{L}$ is a finite set $\{s_1 = t_1, \ldots, s_n = t_n\}$ where s_i, t_i are terms based on $\mathcal{L}$.*

An equality set based on $\mathcal{L} + \mathcal{SK}$ is in solved form iff it is the set $\{\Box\}$ or a set of atoms of the form $x = t$ where x is a variable X or a skolem constant sk, t is a term and each such x occurs only once at the left and not at the right. Moreover, if x is a skolem constant then t is not a variable. An equality set in solved form not equal to $\{\Box\}$ is called consistent.

As mentioned before, we extend the notion of substitution to allow skolem constants to appear in the domain. Application of a substitution on terms and composition of substitutions are defined as in [LMM88, Llo87] and in section 2.1 but by treating skolem constants as variables. With a consistent equality set E in solved form, a unique substitution σ corresponds. This substitution is idempotent: the variables of $dom(\sigma)$ do not occur in the right-hand terms; as a consequence, for each term t, $\sigma(\sigma(t)) = \sigma(t)$. In the sequel we will treat the notions of consistent equality set and substitution as identical, writing for example $\{X = a\}(f(X, a, Y))$ and $E(f(X, a, Y))$ to denote $\{X/a\}(f(X, a, Y))$ and $\sigma(f(X, a, Y))$, respectively. Vice versa, σ will be occasionally used to denote the corresponding equality set E. Concepts as unifier, "*... is more general than ...*" and most general unifier are defined as in [LMM88], again by treating skolem constants as variables. The empty substitution is denoted by ε.

Definition 5.2.4 *An equality set E_s is a solved form of an equality set E iff E_s is in solved form and E_s is an mgu of E or, if no mgu exists, E_s is $\{\Box\}$.*

To unify a set of terms including skolem constants, the unification algorithm of [MM82] must be slightly extended. The modified algorithm treats skolem constants as variables.

Definition 5.2.5 *The* equality reduction *is the process of transforming a set E of equalities to a set E_s of equalities in solved form by applying the following set of rewrite rules (x denotes a variable or a skolem constant):*

(1) $\{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\} \cup E \quad \Rightarrow \quad \{t_1 = s_1, \ldots, t_n = s_n\} \cup E$

(2) $\{f(t_1, \ldots, t_m) = g(s_1, \ldots, s_n\} \cup E \quad \Rightarrow \quad \{\Box\}$ *(failure)*
where $f/m \not\equiv g/n$

(3) $\{x = x\} \cup E \quad \Rightarrow \quad E$

(4) $\{t = x\} \cup E \quad \Rightarrow \quad \{x = t\} \cup E$
where either x is a variable and t is not or
x is a skolem constant and t is neither a variable nor a skolem

(5) $\{x = t\} \cup E \quad \Rightarrow \quad \{\Box\}$ *(failure)*
where $x \not\equiv t$ *and* x *appears in* t *(occur check)*

(6) $\{x = t\} \cup E \quad \Rightarrow \quad \{x = t\} \cup \{x/t\}(E)$
where $x \not\equiv t$, x *appears in* E *and not in* t *and*
if x *is a skolem constant then* t *is not a variable.*

If E *reduces to* $\{\Box\}$, *we say that the equality reduction fails, otherwise it succeeds.*

In section 5.4.1, we prove that equality reduction returns a solved form of E.

Example Consider the following equality set:

$$\{f(X, h(sk, Z), h(sk, sk)) = f(g(Y), Y, Y)\}$$

It has the following equality reduction:

$\{\underline{f(X, h(sk, Z), h(sk, sk)) = f(g(Y), Y, Y)}\}$	rule (1)
$\{\underline{X = g(Y)}, \underline{h(sk, Z) = Y}, h(sk, sk)) = Y\}$	rule (4)
$\{X = g(Y), \underline{Y = h(sk, Z)}, h(sk, sk)) = Y\}$	rule (6)
$\{X = g(h(sk, Z)), Y = h(sk, Z), \underline{h(sk, sk)) = h(sk, Z)}\}$	rule (1)
$\{X = g(h(sk, Z)), Y = h(sk, Z), \underline{sk = sk}, sk = Z\}$	rule (3)
$\{X = g(h(sk, Z)), Y = h(sk, Z), \underline{sk = Z}\}$	rule (4)
$\{X = g(h(sk, Z)), Y = h(sk, Z), \underline{Z = sk}\}$	rule (6)
$\{X = g(h(sk, sk)), Y = h(sk, sk), Z = sk\}$	

Using the definition of equality reduction, we define the extended unification, called *positive unification*. Recall that $\overline{X}$, $\overline{s}$, $\overline{t}$ denote vectors of variables $(X_1, \ldots, X_n)$ and of terms $(s_1, \ldots, s_n), (t_1, \ldots, t_n)$ respectively. An expression $\overline{s} = \overline{t}$ denotes the equality set $\{s_1 = t_1, \ldots, s_n = t_n\}$.

Definition 5.2.6 (positive unification) *Given is an equality set* E *with a consistent solved form* E_s. *Let* θ_{sk} *be a skolemising substitution for the terms which are assigned to skolem constants in* E_s. *A* positive unifier *of* E *is given by the substitution* $\theta_{sk} o E_s$. *A positive unifier of atoms* $p(\overline{t}), p(\overline{s})$ *is a positive unifier of the equality set* $\overline{t} = \overline{s}$.

Based on positive unification, we can define *positive resolution.*

Definition 5.2.7 (positive resolution) *Let* Q *be a goal* $\leftarrow L_1, \ldots, L_m, \ldots, L_k$, *with* L_m *an atom and let* C *be a normal clause* $A \leftarrow B_1, \ldots, B_q$ *sharing no variables with* Q. Q' *is derived from* Q *and* C *by positive resolution on* L_m *and using a positive unifier* θ *if the following conditions hold:*

- θ *is a positive unifier of* L_m *and* A.

- Q' *is the goal* $\theta(\leftarrow L_1, \ldots, L_{m-1}, B_1, \ldots, B_q, L_{m+1}, \ldots, L_k)$.

So far, the procedures that we introduced, can be found elsewhere in the literature. The procedures defined below for negative goals are new.

The case where an abducible atom A is selected in a negative goal is more complex than the positive case. We must compute the failure tree obtained by resolving the goal, in the sequel called a *negative abductive goal*, with all abduced atoms in Δ. The main problem is that the final Δ may not be totally known when the abductive goal is selected. We illustrate the problem with an example. Consider the program with abducible predicate $r/1$:

$q \leftarrow r(X), \neg p(X)$
$p(X) \leftarrow r(b)$

Below, an SLDNFA refutation for the query $\leftarrow r(a), \neg q$ is given. $\mathcal{PG}$ and $\mathcal{NG}$ denote respectively the sets of positive and negative goals. The selected atom at each step is underlined. Only the modified sets $\mathcal{PG}, \mathcal{NG}$ and Δ at each step are given. Initially $\mathcal{NG}$ and Δ are empty.

$\mathcal{PG} = \{\leftarrow \underline{r(a)}, \neg q\}$	*Abduction*
$\mathcal{PG} = \{\leftarrow \underline{\neg q}\}$, $\Delta = \{r(a)\}$	*Switch to* $\mathcal{NG}$
$\mathcal{PG} = \{\Box\}$, $\mathcal{NG} = \{\leftarrow \underline{q}\}$	*Negative resolution*
$\mathcal{NG} = \{\leftarrow \underline{r(X)}, \neg p(X)\}$	*Selection of abducible atom*

If $r/1$ was a defined predicate then at this point we should resolve the selected goal with each clause of the definition of $r/1$. Instead, we are computing a definition for $r/1$ in Δ. Therefore, the atom $r(X)$ must be resolved with all facts already abduced or to be abduced about $r/1$. The problem now is that the set $\{r(a)\}$ is incomplete: indeed, it is easy to see that the resolution of the goal with $r(a)$ will ultimately lead to the abduction of $r(b)$. Hence, the failure tree cannot be computed completely at this point of the computation.

The procedures of [Sha89] and [MBD92], solve this problem by storing all negative literals for which a failure tree is to be computed and rebuilding their failure trees each time a new fact is abduced. As indicated by the authors, this may introduce a serious overhead. SLDNFA avoids this by interleaving the computation of this failure tree with the construction of Δ. This is implemented by storing for each negative abductive goal the triplet (Q, A_Q, D_Q) where Q is the negative abductive goal, A_Q is the abducible atom selected in Q and D_Q is the set of abduced atoms which have already been resolved with Q. $\mathcal{NAG}$ will denote the set of all such triplets. We illustrate this strategy on the example. Initially $\mathcal{NAG}$ is empty. At the current point in the computation, the only abduced fact that can be resolved with the selected goal is $r(a)$. The triplet (" $\leftarrow r(X), \neg p(X)$", "$r(X)$", $\{r(a)\}$) is saved in $\mathcal{NAG}$ and the resolvent $\leftarrow \neg p(a)$ is added to $\mathcal{NG}$:

$\mathcal{NG} = \{\leftarrow \underline{\neg p(a)}\}$, $\mathcal{NAG} = \{(\text{" } \leftarrow r(X), \neg p(X)\text{"}, \text{"}r(X)\text{"}, \{r(a)\})\}$
Switch to $\mathcal{PG}$
$\mathcal{PG} = \{\Box\ ,\ \leftarrow \underline{p(a)}\}$, $\mathcal{NG} = \{\}$ *Positive resolution*
$\mathcal{PG} = \{\Box\ ,\ \leftarrow \underline{r(b)}\}$ *Abduction*
$\mathcal{PG} = \{\Box\}$, $\Delta = \{r(a), \underline{r(b)}\}$ $\mathcal{NAG}$ *goal selected*

Due to the abduction of $r(b)$, another branch starting from the goal in $\mathcal{NAG}$ has to be explored:

$\mathcal{NG} = \{\leftarrow \underline{\neg p(b)}\}$, $\mathcal{NAG} = \{(\text{" } \leftarrow r(X), \neg p(X)\text{"}, \text{"}r(X)\text{"}, \{r(a), r(b)\})\}$
Switch to $\mathcal{PG}$
$\mathcal{PG} = \{\Box\ ,\ \leftarrow \underline{p(b)}\}$, $\mathcal{NG} = \{\}$ *Positive resolution*
$\mathcal{PG} = \{\Box\ ,\ \leftarrow \underline{r(b)}\}$ *Abduction*
$\mathcal{PG} = \{\Box\}$

At this point, a solution is obtained: all positive goals are reduced to $\Box$, the set of negative goals is empty and with respect to Δ, a complete failure tree has been constructed for the negative abductive goal in $\mathcal{NAG}$.

The occurrence of skolem constants in negative goals causes additional problems. The following example illustrates them. Consider the clause:

$$p(f(g(Z), V)) \leftarrow q(Z, V)$$

and the execution of the query $\leftarrow r(X), \neg p(f(X, a))$, where, again, $r/1$ is abducible:

$\mathcal{PG} = \{\leftarrow \underline{r(X)}, \neg p(f(X, a))\}$ *Abduction*
$\mathcal{PG} = \{\leftarrow \underline{\neg p(f(sk, a))}\}$, $\Delta = \{r(sk)\}$ *Switch to* $\mathcal{NG}$
$\mathcal{PG} = \{\Box\}$, $\mathcal{NG} = \{\leftarrow \underline{p(f(sk, a))}\}$ *Negative resolution*

To solve the negative goal $\leftarrow p(f(sk, a))$, we must unify the terms $f(sk, a))$ and $f(g(Z), V)$. Here V and a unify as in normal unification. If we make the default assumption that sk is different from $g(Z)$ for each Z, then the unification fails and therefore $\leftarrow p(f(sk, a))$ fails. However, in general sk may appear in other goals and may be unified there with other terms at a later stage. Assume that due to some unification, sk is assigned a term $g(t)$. In that case, we must retract the default assumption and investigate the new negative goal $\leftarrow q(t, a)$. Otherwise, if all other goals have been refuted, we can conclude the SLDNFA-refutation as a whole by returning $sk \neq g(Z)$ as a constraint on the generated solution. As we will show later on, adding these constraints explicitly is not even necessary.

SLDNFA's *negative unification procedure* obtains this behaviour as follows. First the equality reduction is applied on $f(sk, a)) = f(g(Z), V)$, producing $\{V = a, sk = g(Z)\}$. The variable part $\{V = a\}$ is applied as in normal resolution.

The skolem part $\{sk = g(Z)\}$, which contains the negation of the default assumption, is added as a residual atom to the resolvent and the resulting resolvent $\leftarrow sk = g(Z), q(Z,a)$ is added to $\mathcal{NG}$. The selection of the entire goal can be delayed as long as no value is assigned to sk. If such an assignment occurs and for example the term $g(t)$ is assigned to sk, then the goal $\leftarrow g(t) = g(Z), q(Z,a)$ reduces to the negative goal $\leftarrow q(t,a)$ which then needs further investigation. Otherwise, no further refutation is needed.

This extension of unification and resolution for negative goals is formalised in the following definitions.

Definition 5.2.8 (negative unification) *Given is an equality set E with a consistent solved form E_s. Let θ be the part of E_s with variables at the left and E_r the part of E_s with skolems at the left. We say that E negatively unifies with substitution θ and residue E_r. Two atoms $p(\overline{t}), p(\overline{s})$ negatively unify (with substitution θ and residue E_r) if $\overline{t} = \overline{s}$ unifies (with substitution θ and residue E_r).*

Definition 5.2.9 *We say that $s = t$ is irreducible when s is a skolem constant and t is a non-variable term, different from s.*

An irreducible equality atom $sk = t$ in a negative goal can be used as the default assumption that sk and t are different.

Based on negative unification, we define *negative resolution.*

Definition 5.2.10 (negative resolution) *Let Q be $\leftarrow L_1, \ldots, L_m, \ldots, L_k$, with L_m an atom and let C be a normal clause $A \leftarrow B_1, \ldots, B_q$ sharing no variables with Q.*

Q' is derived from Q and C by negative resolution on L_m if the following holds:

- *L_m and A negatively unify with variable substitution θ and residue:*

$$\{sk_1 = s_1, \ldots, sk_l = s_l\}$$

- *Q' is the goal:*

$$\theta(\leftarrow L_1, \ldots, L_{m-1}, sk_1 = s_1, \ldots, sk_l = s_l, B_1, \ldots, B_q, L_{m+1}, \ldots, L_k)$$

The following property is obvious.

Proposition 5.2.1 *If the selected atom L_m does not contain skolem constants, then positive and negative resolution collapse to classical resolution.*

5.3 The SLDNFA procedure

Below we assume that a normal logic program P based on a language $\mathcal{L}$ with variables $\mathcal{V}$ is given. $\mathcal{SK}$ is a set of skolem constants such that $\mathcal{SK} \cap \mathcal{L} = \phi$ and $\mathcal{D}$ is a skolemisation mapping from $\mathcal{V}$ to $\mathcal{SK}$.

An SLDNFA-derivation is a sequence of quadruples $(\mathcal{PG}, \mathcal{NG}, \mathcal{NAG}, \Delta)$ of multisets of goals and sets of abduced atoms, in which each quadruple is obtained from the previous by applying some SLDNFA-inference operator. Here $\mathcal{PG}$ is the multiset of positive goals; $\mathcal{NG}$ is the multiset of negative goals; Δ is the set of abduced atoms. $\mathcal{NAG}$ is a multiset of triplets (Q, A_Q, D_Q), where Q is a negative abductive goal, A_Q is the abducible atom selected in Q and D_Q is the subset of atoms of Δ which have been resolved with A_Q. Each SLDNFA inference step is initiated by making a selection from one of these (multi-)sets.

Definition 5.3.1 (Selection) *Given a quadruple $(\mathcal{PG}, \mathcal{NG}, \mathcal{NAG}, \Delta)$, a selection is either a tuple (Q, L_m) where $Q \in \mathcal{PG}$ or $Q \in \mathcal{NG}$ and L_m is a literal in Q, or a tuple $((Q, A_Q, D_Q), B)$ where $(Q, A_Q, D_Q) \in \mathcal{NAG}$ and $B \in \Delta$ such that B is negatively unifiable with A_Q and occurs in $\Delta \setminus D_Q$.*

Given a quadruple $(\mathcal{PG}, \mathcal{NG}, \mathcal{NAG}, \Delta)$ and a selection in it, one or more primitive SLDNFA-inference operators can be applied on it. Each operator computes a new tuple $(\mathcal{PG}', \mathcal{NG}', \mathcal{NAG}', \Delta')$ and a substitution θ. The operator deletes the selected goal Q or selected tuple (Q, A_Q, D_Q) from the corresponding multiset, produces a substitution θ and zero, one or more new positive goals, negative goals, triplets with negative abductive goal, selected abducible atom and set of abduced atoms. These expressions are added to the corresponding (multi)-sets and the substitution θ is applied on all (multi-)sets to obtain the new tuple $(\mathcal{PG}', \mathcal{NG}', \mathcal{NAG}', \Delta')$.

Abduction, positive and negative resolution are the main primitive inference operators in SLDNFA. There are two negative resolution operators: one for negative goals and one for negative abductive goals. Three other operators deal with negative literals in positive and negative goals. Finally, one operator is applied when an abducible atom is selected in a negative goal. This operator merely moves the negative goal from $\mathcal{NG}$ to $\mathcal{NAG}$ and initialises D_Q to ϕ. Below we formalise each operator.

Definition 5.3.2 (positive resolution operator) *The* positive resolution operator *applies when a defined atom A or an equality atom $s = t$ is selected in a positive goal.*

If a defined atom is selected, then let Q' be derived from Q and a variant of a program clause of P by positive resolution on A and using the positive unifier θ. If an equality atom $s = t$ is selected, then let Q' be derived from Q and a variant of the reflexivity atom $X = X \leftarrow$ by positive resolution on $s = t$ and using a

positive unifier θ. The positive resolution operator *produces the substitution θ and the positive goal Q' . Formally:*

$$\mathcal{PG}' = \theta(\mathcal{PG} \setminus \{Q\}) \cup \{Q'\}$$
$$\mathcal{NG}' = \theta(\mathcal{NG}),\ \mathcal{NAG}' = \theta(\mathcal{NAG}) \text{ and } \Delta' = \theta(\Delta)$$

Definition 5.3.3 (abduction operator) *The* abduction operator *applies when an abducible atom A is selected in a positive goal Q.*

Let Q' be derived from Q by abducing A using the skolemising substitution θ. The abduction operator produces the substitution θ, the abduced atom $\theta(A)$ and the positive goal Q'. Formally:

$$\mathcal{PG}' = \theta(\mathcal{PG} \setminus \{Q\}) \cup \{Q'\}$$
$$\Delta' = \theta(\Delta) \cup \{\theta(A)\}$$
$$\mathcal{NAG}' = \theta(\mathcal{NAG}), \mathcal{NG}' = \theta(\mathcal{NG})$$

Two goals in $\mathcal{PG}_i \cup \mathcal{NG}_i \cup \mathcal{NAG}_i$ do not share variables, only skolem constants can be shared. Therefore, only the skolem part of a substitution θ can have an effect when applying θ on other goals or abduced atoms. Since the substitution θ generated by the abduction operator is a variable substitution, application on other goals has no effect.

Definition 5.3.4 (switch to $\mathcal{NG}$ operator) *The* switch to $\mathcal{NG}$ operator *applies when a negative literal $\neg A$ is selected in a positive goal Q.*

Let Q' be obtained from Q by deleting $\neg A$. The switch to $\mathcal{NG}$ operator *produces the empty substitution, the negative goal $\leftarrow A$ and the positive goal Q'. Formally:*

$$\mathcal{PG}' = \mathcal{PG} \setminus \{Q\} \cup \{Q'\}$$
$$\mathcal{NG}' = \mathcal{NG} \cup \{\leftarrow A\}$$
$$\mathcal{NAG}' = \mathcal{NAG}, \Delta' = \Delta \text{ and } \theta = \varepsilon$$

Definition 5.3.5 (negative resolution operator) *The* negative resolution operator *applies when a defined atom A or an equality atom $s = t$ is selected in a negative goal Q.*

If a defined atom is selected, then let S be the set of all resolvents that can be derived by negative resolution on A from Q and precisely one variant with fresh variables for each clause of P. If an equality atom $s = t$ is selected, then let S be the singleton $\{Q'\}$ where Q' is derived from Q and a variant of the reflexivity atom $X = X \leftarrow$ by negative resolution on $s = t$. The negative resolution operator *produces the empty substitution and the set of negative goals S. Formally:*

$$\mathcal{NG}' = \mathcal{NG} \setminus \{Q\} \cup S$$
$$\mathcal{PG}' = \mathcal{PG}, \mathcal{NAG}' = \mathcal{NAG}, \Delta' = \Delta \text{ and } \theta = \varepsilon$$

Note that when S is empty, the result of the operation is to delete Q from $\mathcal{NG}$.

A smart selection rule will never select a negative goal which contains an irreducible equality atom $sk = t$ because it will fail anyway if, eventually, the default assumption $sk \neq t$ is added as a constraint on the solution. One easily verifies that applying the negative resolution operator on a selection of a negative goal and an irreducible equality atom has no effect. This is because negative unification of $sk = t$ and $X = X$ returns the empty substitution and residue $\{sk = t\}$. As a consequence, selecting such a goal and atom may lead to a loop.

Definition 5.3.6 (move to $\mathcal{NAG}$ operator) *The* move to $\mathcal{NAG}$ operator *applies when an abducible atom A is selected in a negative goal Q.*

The move to $\mathcal{NAG}$ operator *produces the empty substitution and the tuple* $(Q, A, \{\})$. *Formally:*

$$\mathcal{NG}' = \mathcal{PG} \setminus \{Q\}$$
$$\mathcal{NAG}' = \mathcal{NAG} \cup \{(Q, A, \{\})\}$$
$$\mathcal{PG}' = \mathcal{PG},\ \Delta' = \Delta \text{ and } \theta = \varepsilon$$

Definition 5.3.7 (switch to $\mathcal{PG}$ and failed negation operator) *The* switch to $\mathcal{PG}$ operator *and* failed negation operator *are both applicable when a tuple* $(Q, \neg A)$ *is selected such that* $Q \in \mathcal{NG}$.

The switch to $\mathcal{PG}$ operator *produces the empty substitution and the positive goal* $\leftarrow A$. *Formally:*

$$\mathcal{PG}' = \mathcal{PG} \cup \{\leftarrow A\}$$
$$\mathcal{NG}' = \mathcal{NG} \setminus \{Q\}$$
$$\mathcal{NAG}' = \mathcal{NG},\ \Delta' = \Delta \text{ and } \theta = \varepsilon$$

Let Q' be obtained from Q by deleting $\neg A$. The failed negation operator *operator produces the empty substitution and the negative goals $\leftarrow A$ and Q'. Formally:*

$$\mathcal{NG}' = \mathcal{NG} \setminus \{Q\} \cup \{Q', \leftarrow A\ \}$$
$$\mathcal{PG}' = \mathcal{PG}, \mathcal{NAG}' = \mathcal{NAG},\ \Delta' = \Delta \text{ and } \theta = \varepsilon$$

During execution of SLDNFA, the selection of a negative literal in a negative goal creates a backtracking point: first one operator is applied, after backtracking the second is applied.

The intuition behind the two operators is the following. SLDNFA tries to fail the negative goal $Q = \leftarrow L_1, \ldots, \neg A, \ldots, L_n$, i.e. it tries to prove Q or equivalently $\forall(\neg L_1 \vee \ldots \vee \neg\neg A \vee \ldots \vee \neg L_n)$. Let Q' be the goal obtained by deleting $\neg A$ from Q. The following equivalences hold:

$$\begin{aligned} Q &\Leftrightarrow \forall(\neg L_1 \vee \ldots \vee A \vee \ldots \vee \neg L_n) \\ &\Leftrightarrow A \vee Q' \\ &\Leftrightarrow A \vee (\neg A \wedge Q') \end{aligned}$$

Because A is ground, A can be moved outside the universal quantifiers of Q. The second transition follows from the tautology $p \vee q \Leftrightarrow p \vee (\neg p \wedge q)$. The switch to $\mathcal{PG}$ operator tries to prove Q by building a successful derivation for $\leftarrow A$, thus proving A. The failed negation operator tries to prove Q by finitely failing $\leftarrow A$ and Q', thus proving $\neg A \wedge Q'$. An alternative for the failed negation operator would be to add only Q'. The correctness of this simplified operator follows from the simpler equivalence $Q \Leftrightarrow A \vee Q'$. This approach is followed in the procedure defined in [KM90a]. However, this variant appears to be in general less efficient, due to the fact that redundant solutions in both branches can be explored: solutions constructed via the weaker failed negation operator may satisfy A and thus are investigated when applying the switch to $\mathcal{PG}$ operator. The failed negation operator avoids this by implementing a form of *complement splitting*.

Definition 5.3.8 ($\mathcal{NAG}$ operator) *The* $\mathcal{NAG}$ operator *applies when a tuple of the form* $((Q, A_Q, D_Q), B)$ *is selected such that* Q *is a negative abductive goal and* B *an abduced fact from* $\Delta \setminus D_Q$.

Let Q' *be the resolvent derived from* Q *and* B *by negative resolution on* A_Q. *The* $\mathcal{NAG}$ *operator produces the empty substitution, the negative goal* Q' *and the tuple* $(Q, A_Q, D_Q \cup \{B\})$. *Formally:*

$$\mathcal{NG}' = \mathcal{NG} \cup \{Q'\}$$
$$\mathcal{NAG}' = \mathcal{NAG} \setminus \{(Q, A_Q, D_Q)\} \cup \{(Q, A_Q, D_Q \cup \{B\})\}$$
$$\mathcal{PG}' = \mathcal{PG},\ \Delta' = \Delta \text{ and } \theta = \varepsilon$$

Definition 5.3.9 (Safe selection) *A selection is safe when it is not a tuple* $(Q, \neg A)$ *with* A *a non-ground atom. Skolem constants may appear in* A.

Definition 5.3.10 *Let* P *be a normal program based on a language* $\mathcal{L}$ *with variables* $\mathcal{V}$. *Let* $\mathcal{SK}$ *be a set of skolem constants such that* $\mathcal{SK} \cap \mathcal{L} = \phi$ *and let* $\mathcal{D}$ *be a skolemisation mapping from* $\mathcal{V}$ *to* $\mathcal{SK}$.

An SLDNFA derivation for a query Q_0 *consists of finite or infinite sequences:*

- $\{Q_0\} = \mathcal{PG}_0, \mathcal{PG}_1, \ldots$ *of multisets of goals,*
- $\{\} = \mathcal{NG}_0, \mathcal{NG}_1, \ldots$ *of multisets of goals,*
- $\{\} = \mathcal{NAG}_0, \mathcal{NAG}_1, \ldots$ *of multisets of triplets* (Q, A_Q, D_Q), *where* Q *is a goal,* A_Q *is an abducible atom in* Q *and* D_Q *is a set of ground atoms, negatively unifiable with* A_Q,
- $\{\} = \Delta_0, \Delta_1, \ldots$ *of sets of ground abducible facts, and*
- $\theta_1, \theta_2, \ldots$ *of skolem substitutions.*

Moreover, for each i, *there is a safe selection in* $(\mathcal{PG}_i, \mathcal{NG}_i, \mathcal{NAG}_i, \Delta_i)$, *and* $(\mathcal{PG}_{i+1}, \mathcal{NG}_{i+1}, \mathcal{NAG}_{i+1}, \Delta_{i+1})$ *and the substitution* θ_{i+1} *are obtained by applying one SLDNFA-inference operator on the selection.*

The set of skolem constants occurring in Δ is denoted by $Sk(\Delta)$.

We require the selection in an SLDNFA-derivation to be safe: a negative literal may only be selected when L_m is ground. Observe that because the selection is safe, two goals in $\mathcal{PG}_i \cup \mathcal{NG}_i \cup \mathcal{NAG}_i$ do not share variables, only skolem constants can be shared.

Definition 5.3.11 *An SLDNFA-derivation is finitely failed if it is finite, say of length n, and one of the following situations occurs at n:*

- *a defined atom A is selected in a positive goal $Q \in \mathcal{PG}_n$ such that no positive resolution is possible on A.*
- $\square \in \mathcal{NG}_n$.

Definition 5.3.12 *An SLDNFA refutation K for a goal Q is a finite SLDNFA derivation (say of length n) such that $\mathcal{PG}_n$ contains no other goals than $\square$, each goal Q in $\mathcal{NG}_n$ comprises an irreducible equality atom and for each (Q, A_Q, D_Q) in $\mathcal{NAG}_n$: D_Q contains each $B \in \Delta_n$ which negatively unifies with A_Q.*

The answer substitution is $\theta_a = \theta_n o \ldots o \theta_1|_{var(Q)}$. The solution generated by K is (Δ_n, θ_a).

Example SLDNFA is applied to a small fault diagnosis problem on a lamp. A faulty lamp problem is caused by a broken lamp or by a power failure of a circuit without backup. The circuit $c1$ is equipped with a battery $b1$ as backup. The battery can be empty. Formalised:

$lamp(l1)$:-
$battery(c1, b1)$:-
$faulty\_lamp$:- $lamp(X), broken(X)$
$faulty\_lamp$:- $powerfailure(X), \neg backup(X)$
$backup(X)$:- $battery(X, Y), \neg empty(Y)$

The predicates $broken/1$, $powerfailure/1$ and $empty/1$ are undefined. An SLDNFA derivation for the goal $\leftarrow faulty\_lamp$ is:

$\mathcal{PG} = \{\leftarrow \underline{faulty\_lamp}\}$	*Positive resolution*
$\mathcal{PG} = \{\leftarrow \underline{powerfailure(X)}, \neg backup(X)\}$	*Abduction*
$\mathcal{PG} = \{\leftarrow \underline{\neg backup(sk)}\},\ \Delta = \{powerfailure(sk)\}$	*Switch to $\mathcal{NG}$*
$\mathcal{PG} = \{\square\},\ \mathcal{NG} = \{\leftarrow \underline{backup(sk)}\},$	*Negative resolution*
$\mathcal{NG} = \{\leftarrow \underline{battery(sk, Y)}, \neg empty(Y)\},$	*Negative resolution*
$NG = \{\leftarrow \underline{sk = c1}, \neg empty(b1)\}$	

The derivation can terminate here, generating $\Delta = \{powerfailure(sk)\}$ and $\mathcal{NG} = \{\leftarrow \underline{sk = c1}, \neg empty(b1)\}$, representing the solution that there is a

power failure on a circuit sk which is not $c1$. Note that the solution with $c1$ and an empty battery $b1$ is not derived and cannot be derived by any SLDNFA derivation. We return to this problem in section 5.7.

An example of a failed derivation is obtained for the following query:

$$\leftarrow \neg broken(l1), faulty\_lamp$$

The failed derivation goes as follows:

$$\begin{array}{ll}
\mathcal{PG} = \{\leftarrow \underline{\neg broken(l1)}, faulty_lamp\} & \textit{Switch to } \mathcal{NG} \\
\mathcal{PG} = \{\leftarrow \underline{faulty_lamp}\},\ \mathcal{NG} = \{\leftarrow \underline{broken(l1)}\} & \textit{Move to } \mathcal{NAG} \\
\mathcal{PG} = \{\leftarrow \underline{faulty_lamp}\},\ \mathcal{NG} = \{\}, & \\
\qquad \mathcal{NAG} = \{(\text{" } \leftarrow broken(l1)\text{"}, \text{"}broken(l1)\text{"}, \phi)\} & \\
 & \textit{Positive resolution} \\
\mathcal{PG} = \{\leftarrow \underline{lamp(X)}, broken(X)\} & \textit{Positive Resolution} \\
\mathcal{PG} = \{\leftarrow \underline{broken(l1)}\} & \textit{Abduction} \\
\mathcal{PG} = \{\Box\},\ \Delta = \{\underline{broken(l1)}\}, & \\
\qquad \mathcal{NAG} = \{\underline{(\text{" } \leftarrow broken(l1)\text{"}, \text{"}broken(l1)\text{"}, \phi)}\} & \\
 & \textit{NAG operation} \\
\mathcal{NG} = \{\Box\} &
\end{array}$$

At this point the derivation fails and computation backtracks and selects the second clause of $faulty\_lamp$.

A refutation generates not only abduced atoms but also constraints in $\mathcal{NG}_n$ of the form
$\leftarrow L_1, \ldots, sk = t, \ldots, L_k$. These constraints are valid under the default assumption that $sk \neq t$. Clearly this assumption is satisfied in the theory $FEQ(\mathcal{L} + Sk(\Delta_n))$. Therefore $FEQ(\mathcal{L} + Sk(\Delta_n)) \models \leftarrow L_1, \ldots, sk = t, \ldots, L_k$. Thus, we do not need to add these constraints explicitly to the solution after all, as announced earlier.

Another issue is the relation to SLDNF. Although the definition of SLDNFA refutation is structured rather differently than the definition of SLDNF refutation [Llo87], for complete logic programs, the definitions of SLDNFA and SLDNF are equivalent: for every SLDNFA refutation, it is possible to construct an SLDNF refutation with the same goals, resolution steps and substitutions. Vice versa, for every SLDNF refutation an equivalent SLDNFA refutation can be constructed.

Proposition 5.3.1 *For each SLDNF refutation for a goal Q and complete program P, there exists an SLDNFA refutation generating the same answer substitution. Vice versa, for each SLDNFA refutation, there exists an SLDNF refutation generating the same answer substitution.*

Proof We only sketch the proof.

Assume that there exists an SLDNFA refutation, say of length n, for a query Q_0. Consider the set $\mathcal{G} = \{Q_{k_i} \mid i \geq 0\}$ consisting of Q_0 and all ground atomic goals $\leftarrow A$ created by the *switch to* $\mathcal{PG}$, *switch to* $\mathcal{NG}$ and *failed negation* operations upon selection of a tuple $(Q, \neg A)$. With each positive goal in $\mathcal{G}$, a sequence of positive goals can be associated, each of which is obtained from the previous by positive resolution or by the elimination of a negative literal at *a switch to* $\mathcal{NG}$ operation. Due to the fact that, without skolem constants, positive resolution collapses to classical resolution, this sequence of goals forms an SLDNF derivation.

With each negative goal in $\mathcal{G}$, similarly a SLDNF failure-tree of goals can be associated. The descendants of a node in which a positive literal has been selected, are the resolvents obtained by applying the *negative resolution* operator. For a node in which a negative literal $\neg A$ was selected, there are two possibilities. Either the positive goal $\leftarrow A$ was added in the SLDNFA refutation by the *switch to* $\mathcal{PG}$ operator (and an SLDNF derivation exists for this goal). In that case, the node has no descendants; or, the *failing negation* operator was applied on the goal and $\leftarrow A$ was added as a negative goal (and an SLDNF-failure tree exists for it). In that case, the goal has as descendant the remainder of the negative goal.

Since an SLDNFA-refutation is finite, each SLDNF derivation sequence and each failure tree is finite. Each SLDNF derivation depends on the set of failure trees of negative goals introduced by *switch to* $\mathcal{NG}$ operations. Vice versa, an SLDNF failure tree depends on the SLDNF derivations of the ground atomic positive goals produced by *switch to* $\mathcal{PG}$ operations and on the failure trees for ground atomic negative goals introduced by *failing negation* operations. Since the SLDNFA refutation is finite, the SLDNF derivations and failure trees can be correctly attributed a finite rank. Thus, we obtain an SLDNF refutation with the rank of the derivation sequence of Q_0.

Let $\theta_{i_1}, \ldots, \theta_{i_l}$ be the sequence of substitutions for the SLDNF refutation for Q_0. It is a subsequence of $\theta_1, \ldots, \theta_n$. Two different SLDNF derivations or failure trees do not share variables, so that the substitutions used in one derivation do not affect other derivations or failure trees. Therefore, the answer substitution $\theta_{i_l} o \ldots o \theta_{i_1}|_{var(Q_0)}$ of the SLDNF refutation is identical to the answer substitution $\theta_n o \ldots o \theta_1|_{var(Q_0)}$ for the SLDNFA refutation.

Second, assume that some SLDNF refutation of rank k for Q_0 exists. By reverse engineering on the first part of this proof, it is easily verified that an SLDNFA refutation can be constructed from the SLDNF refutations and failure trees. □

In the next sections we will prove the soundness and completeness of SLDNFA

wrt 3-valued completion semantics. 3-valued completion is a better choice than 2-valued completion semantics: it gives better results for problematic programs containing clauses like p :- $\neg p$. It is also a *fair* choice: completion semantics or direct justification semantics is the weakest semantics of the three families of semantics considered in chapter 4 for which SLDNFA is sound. Here the *weakest* semantics means the semantics with most models and fewest implied formulas.

The soundness result expresses that for a computed answer (Δ, θ) for a query $\leftarrow L_1, \ldots, L_k$ in a program P, the completion $comp_3(P + \Delta)$ of the complete logic program $P + \Delta$ is consistent and implies $\forall(\theta(L_1 \wedge \ldots \wedge L_k))$. Note that by theorem 4.3.3, this implies the soundness of the procedure wrt (partial) justification semantics. As a completeness result, we will prove that (under certain conditions expressed below), SLDNFA generates all minimal and most general solutions.

Both the soundness and completeness result depend on properties proven in the next section for the primitive inference operators.

5.4 Primitive inference operators

In the sequel, we assume that the abductive normal program P and initial goal Q_0 are based on the (user-defined) language $\mathcal{L}$ and that there exists a skolemisation mapping $\mathcal{D}$ from the variables $\mathcal{V}$ of $\mathcal{L}$ to the set $\mathcal{SK}$ of skolem constants ($\mathcal{SK} \cap \mathcal{L} = \phi$). To deal with the skolem constants which occur in the computed goals, we take as underlying language $\mathcal{L} + \mathcal{SK}$, unless stated explicitly. We denote this language by $\mathcal{L}_{sk}$. For example, when we write $P \models F$, we mean $<\mathcal{L}_{sk}, P> \models F$.

The notation $comp_3(\mathcal{L}, P)$ denotes the theory consisting of $FEQ(\mathcal{L})$ and the if-and-only-if definitions of the predicates of P, expressed via the 3-valued equivalence operator $\Leftrightarrow$. Observe that the underlying language and the language for which FEQ is added are in general not the same: in general the underlying language will be $\mathcal{L}_{sk}$, while FEQ will be constructed for the sub-language $\mathcal{L}$. In fact, an ordinary constant c and a skolem constant sk are logically distinguished precisely by the fact that the constraints of FEQ hold for c and not for sk. If FEQ was defined for the complete language $\mathcal{L}_{sk}$, then skolem substitutions $sk = t$ were always false.

One might wonder what is the effect of adding an infinite number of new constants to the language on the meaning of a FOL theory $\mathcal{T}$. After all, extending $\mathcal{L}$ with $\mathcal{SK}$ has a serious impact on the Herbrand universe. The answer is, sometimes surprisingly, that this has no effect. Indeed, if any model of $<\mathcal{L}, \mathcal{T}>$ is extended by giving each constant of $\mathcal{SK}$ a random domain element as interpretation, then a model of $<\mathcal{L}_{sk}, \mathcal{T}>$ is obtained. Vice versa, by restricting each model of $<\mathcal{L}_{sk}, \mathcal{T}>$ to $\mathcal{L}$, a model of $<\mathcal{L}, \mathcal{T}>$ is obtained[1].

For the proofs of the properties, skolem constants will often be dealt with by replacing them by the original variables. The intuition behind the following lemma

[1] This property is obviously due to the fact that FOL model theory allows general interpretations.

is that if a formula can be proven for some skolem constant, which can represent an arbitrary element of the domain, the formula holds for every element of the domain.

Lemma 5.4.1 (Deskolemisation lemma) *Let $\mathcal{T}$ be a FOL theory and F a closed formula containing skolem constants (which do not occur in $\mathcal{T}$). We assume that no variable and its corresponding constant both occur in F.*
$<\mathcal{L}_{sk}, \mathcal{T}> \models F$ *iff* $<\mathcal{L}, \mathcal{T}> \models \forall(\mathcal{D}^{-1}(F))$

Proof That $<\mathcal{L}, \mathcal{T}> \models \forall(\mathcal{D}^{-1}(F))$ implies $<\mathcal{L}_{sk}, \mathcal{T}> \models F$ is trivial (from the substitution axiom).

Vice versa, assume that $<\mathcal{L}_{sk}, \mathcal{T}> \models F$, and there exists a model M of $<\mathcal{L}, \mathcal{T}>$ and a variable assignment V such that $\mathcal{H}_M(V(\mathcal{D}^{-1}(F))) \neq \mathbf{t}$. As argued higher, any extension of M to $\mathcal{L}_{sk}$ is still a model of $\mathcal{T}$. In particular consider an extension M_V which assigns to each skolem constant sk in F, the value $V(\mathcal{D}^{-1}(sk))$. By its construction, $\mathcal{H}_{M_V}(F) \neq \mathbf{t}$. This contradicts the fact that $<\mathcal{L}_{sk}, \mathcal{T}> \models F$. □

The proof techniques that will be used in most proofs are essentially the ones used in [Cla78]: proofs by rewriting formulas by classical equivalence preserving laws such as applying commutativity, distributivity of $\wedge$ and $\vee$ over each other and over $\exists$ and $\forall$. One aspect that requires some closer attention is that the proofs are for 3-valued logic. This poses only few problems: all classical rewrite rules are $\Leftrightarrow$ -equivalence preserving, i.e. they are based on $\Leftrightarrow$ tautologies (e.g. $F \wedge (G \vee H) \Leftrightarrow (F \wedge G) \vee (F \wedge H)$). These tautologies can easily be proved in a model theoretic way (showing that left and right hand of the equivalence has the same truth value in any 3-valued interpretation). One problem that we do have to circumvent is related to the implication connective $\leftarrow$. Often, we would like to reason as follows: from $F \Leftrightarrow (G \vee H)$, infer $F \leftarrow G$. This is a problematic conclusion: consider a model M in which $\mathcal{H}_M(F) = \mathbf{u} = \mathcal{H}_M(G \vee H) = \mathcal{H}_M(G)$, then we have $\mathcal{H}_M(F \leftarrow G) = \mathbf{u} < \mathcal{H}_M(F \Leftrightarrow (G \vee H))$. Or, even when $F \Leftrightarrow (G \vee H)$ is entailed by a theory, $F \leftarrow G$ is in general not entailed. We avoid the problem by replacing $\leftarrow$ by a 3-valued version of the implication connective, $\Leftarrow$ (and its symmetric version $\Rightarrow$). The truth function of an interpretation M is defined on $\Leftarrow$ as follows:

$$\mathcal{H}_M(F \Leftarrow G) = \mathbf{t} \text{ iff } \mathcal{H}_M(F) \geq \mathcal{H}_M(G)$$
$$\mathcal{H}_M(F \Leftarrow G) = \mathbf{f} \text{ iff } \mathcal{H}_M(F) < \mathcal{H}_M(G)$$

One easily verifies that one may correctly infer $F \Leftarrow G$ from $F \Leftrightarrow (G \vee H)$. Note that the truth function for $\Leftarrow$ is the same as for the connective :- defined in section 2.3. Below we prefer the connectives $\Leftarrow$ and $\Rightarrow$ instead of :- and -:, mainly for aesthetic reasons.

Rewrite rules of special importance are those which mimic the application and the composition of substitutions: for conjunctions we have the tautologies $s = t \wedge F[s] \Leftrightarrow s = t \wedge F[t]$ and $(\exists X : (X = t \wedge F[X])) \Leftrightarrow F[t]$. The latter allows to eliminate variables occurring in the domain of a substitution. Similarly, for disjunctions we have $\neg(s = t) \vee F[s] \Leftrightarrow \neg(s = t) \vee F[t]$ and $(\forall X : (X = t \rightarrow F[X])) \Leftrightarrow F[t]$.

5.4.1 Soundness of Unification

Proposition 5.4.1 (soundness of equality reduction) *The equality reduction applied to a set of equations E will return a solved form E_s of E. Moreover,* $\mathrm{FEQ}(\mathcal{L}) \models \forall(E \Leftrightarrow E_s)$.

Proof The proof of the theorem is analogous to the proof of the correctness of the original algorithm of Martelli and Montanari [MM82, LMM88].

To prove that equality reduction terminates, a well-founded partial order[2] is defined on the set of equation sets such that if E can be rewritten to E' by any rewrite rule of definition 5.2.5 then $E > E'$.

As in [MM82], X is called a solved variable in an equality set E iff $X = t \in E$ and X appears neither in t nor in $E \setminus \{X = t\}$. We call sk a solved skolem constant iff $sk = t \in E$, t is not a variable and sk appears neither in t nor in $E \setminus \{sk = t\}$. With an equality set E, we associate a norm $\|E\|$ which is the tuple (N_1, N_2, N_3, N_4) with:

$N_1 =$ number of unsolved variables and skolem constants in E
$N_2 =$ number of functors at the left of an equation in E
$N_3 =$ number of occurrences of skolem constants at the left of an equation in E
$N_4 =$ number of variable occurrences in E

Now we define for any pair of equality sets E_1, E_2: $E_1 < E_2 \Leftrightarrow \|E_1\| < \|E_2\|$ where "$<$" on quadruples is the lexicographic extension of the standard order on natural numbers. Or $E < E'$ iff $(N_1, N_2, N_3, N_4) < (N_1', N_2', N_3', N_4')$ iff for some $1 \leq i \leq 4 : N_1 = N_1', \ldots, N_{i-1} = N_{i-1}', N_i < N_i'$. This defines a well-founded order on the quadruples and therefore on the equality sets. One easily verifies that for any equality set E, if E can be rewritten to E' with any rewrite rule of definition 5.2.5, then $E' < E$. This proves the termination of the algorithm.

That equality reduction returns a solved form is trivial, since only for a solved form, none of the rewrite rules applies. We prove that (a) equality

[2] A well-founded partial order is a strict partial order in which there exists no infinite decreasing sequence $x_0 > x_1 > \ldots > x_n > \ldots$.

reduction fails iff E has no unifiers and (b) it returns an mgu of the input equality sets iff E has a unifier. The proof is based on the straightforward observation that each rewrite rule preserves all unifiers of the equality sets. As a consequence, all equality sets derived during the unification process have the same unifiers.

((a) $\rightarrow$) Assume that equality reduction fails on E and that E' is the last equality set in the reduction sequence before $\{\Box\}$ is obtained. One easily checks that whatever the rewrite rule is used for rewriting E' to $\{\Box\}$, the selected equation in E' has no unifiers. Hence, since E' and E have the same unifiers, E has no unifier.

((a) $\leftarrow$) Assume that equality reduction does not fail on E and that E can be reduced to a solved form E_s. E and E_s have the same unifiers. Clearly, E_s, interpreted as a substitution, is a unifier of itself and therefore of E.

((b) $\rightarrow$) is trivial, since an mgu is a unifier. We prove ((b) $\leftarrow$). Assume that E has a unifier θ. By (a) $\rightarrow$, it follows that equality reduction does not fail but produces a solved form E_s. We prove that E_s is more general than θ by showing that $\theta = \theta o E_s$. If x is a skolem or variable which does not belong to $dom(E_s)$, then $E_s(x) = x$ and $\theta(E_s(x)) = \theta(x)$. Let $x = t$ be an equality of E_s. We have that $E_s(x) = t$ and $\theta(x) = \theta(t)$, hence $\theta(E_s(x)) = \theta(t) = \theta(x)$. Summarising, we find that for each skolem or variable x, $\theta(E_s(x)) = \theta(t)$. Hence, $\theta = \theta o E_s$.

The proof of the logical equivalence of E and E_s in [Cla78] applies almost without change to the proof of the extended algorithm. One simply observes that each rewrite rule maintains the equivalence wrt $\Leftrightarrow$. Because "=" has a two-valued interpretation, equivalence wrt $\Leftrightarrow$ coincides with equivalence wrt $\Leftrightarrow$ (and the connectives $\leftarrow$ and $\rightarrow$ connectives appearing in *FEQ* coincide with $\Leftarrow$ and $\Rightarrow$). $\Box$

Proposition 5.4.2 *Let E be an equality set.*

(a) $\mathrm{FEQ}(\mathcal{L}) + \exists(E)$ *is consistent iff there exists a consistent solved form E_s of E.*

(b) If E has consistent solved form E_s and M is a model of $\mathrm{FEQ}(\mathcal{L}) + \exists(E)$ *then if V is a variable assignment such that $M \models V(E)$, then $M \models V(E_s)$.*

Proof (a) Assume that $FEQ(\mathcal{L}) + \exists(E)$ is consistent. By Proposition 5.4.1, applying equality reduction on E yields a solved form E_s of E. Since E_s and E are equivalent, E_s must be consistent.

Conversely, assume that E_s is a consistent solved form of E. We have called an equality set in solved form *consistent* if it is not of the form $\{\Box\}$. Due to the equivalence of E_s and E, it suffices to prove that $FEQ(\mathcal{L}) + \exists(E_s)$ is consistent. Let E'_s be the result of deskolemising E_s. E'_s is in solved form.

Since skolemisation preserves satisfiability, $FEQ(\mathcal{L}) + \exists(E_s)$ is consistent iff $FEQ(\mathcal{L}) + \exists(E'_s)$ is consistent.

We construct a model of $FEQ(\mathcal{L}) + \exists(E'_s)$. Consider the Herbrand interpretation $M = \{t = t \| t \in HU(\mathcal{L})\}$. M is clearly a model of $FEQ(\mathcal{L})$. Take some variable assignment V of variables of $range(E'_s)$. Extend it by assigning $V(t_i)$ to each X_i for which $X_i = t_i \in E'_s$. Obviously $M \models V(E'_s)$. That implies $M \models \exists(E'_s)$; i.e. $FEQ(\mathcal{L}) + \exists(E'_s)$ is consistent.

(b) If M is a model of $FEQ(\mathcal{L}) + \exists(E)$ and V is a variable assignment such that $M \models V(E)$, then by the equivalence of E and E_s and the fact the equality reduction does not introduce new variables, $M \models V(E_s)$. If new variables would appear in E_s then $V(E_s)$ would not be ground, and V should be extended to express $M \models V(E_s)$.

□

5.4.2 Soundness of SLDNFA inference operators

In this section we prove for each SLDNFA inference operator a soundness and a completeness result. Each operator can be seen as performing a classical FOL theorem proving step. An SLDNFA inference operator deletes a selected goal Q or selected tuple (Q, A_Q, D_Q) from the corresponding multiset and produces a substitution θ and a set $S = \{F_1, \ldots, F_n\}$ with zero, one or more new positive goals, negative goals, tuples with negative abductive goals and abduced atoms. The soundness result for the operator is described by a formula of the following form:

$$\forall(\theta(\mathcal{M}(Q)) \Leftarrow \mathcal{M}(F_1) \wedge \ldots \wedge \mathcal{M}(F_n))$$

Here $\mathcal{M}(Q), \mathcal{M}(F_1), \ldots, \mathcal{M}(F_n)$ are FOL formulas which are defined in the definition below. Below, for a query $Q = \leftarrow L_1, \ldots, L_n$, the expression $\&(Q)$ denotes the open formula $L_1 \wedge \ldots \wedge L_n$ and $\overline{\forall}(Q)$ denotes the open formula $\neg L_1 \vee \ldots \vee \neg L_n$.

Definition 5.4.1 *The* meaning $\mathcal{M}(Q)$ *of a positive goal Q is the open formula $\&(Q)$. The* meaning $\mathcal{M}(Q)$ *of a negative goal or a negative abductive goal Q is the closed formula $\forall(\overline{\forall}(Q))$, or simply Q*[3]*. The* meaning $\mathcal{M}(A)$ *of an abduced atom A is A itself.*

Example Consider the program P with the following definition for a predicate $r/2$:

$r(a, b)$:-
$r(g(X), g(X))$:- $q(Z)$

[3]Recall that a normal query or goal Q stands for the FOL formula $\forall(\neg L_1 \vee \ldots \vee \neg L_n)$.

and the positive goal $\leftarrow r(sk, V), q(f(sk, V))$ in which $r(sk, V)$ is selected. When the first clause of $r/2$ is selected, the positive resolution operator produces the unifier $\{sk/a, V/b\}$ and the positive goal $\leftarrow q(f(a, b))$. When the second clause is selected, the operator produces the substitution

$$\{X/sk_2\} o \{sk/g(X), V/g(X)\}$$

and the positive goal $\leftarrow q(Z), q(f(g(sk_2), g(sk_2)))$. These two operations can be interpreted as classical theorem proving operations, proving the following implications of $comp_3(P)$:

$$r(a, b) \wedge q(f(a, b)) \Leftarrow q(f(a, b))$$
$$\forall Z : r(g(sk_2), g(sk_2)) \wedge q(f(g(sk_2), g(sk_2))) \Leftarrow q(Z) \wedge q(f(g(sk_2), g(sk_2)))$$

The completeness result is more complex. On a given selection, sometimes only one, sometimes more than one inference operation can be applied. For example, given a selection of a defined atom in a positive goal, the positive resolution operator can be applied for each program clause unifiable with the selected atom. Given a selection of a negative literal in a negative goal, the switch to $\mathcal{PG}$ and the failed negation operators can be applied. On all other selections, only one operation is possible. If different operations are possible, backtracking is necessary.

Assume that for a selected goal Q or tuple (Q, A_Q, D_Q), a number of inference operations can be applied, generating substitutions $\theta_1, \ldots, \theta_k$ and sets $S_1, \ldots, S_k$ of produced expressions. The completeness result for each selection is described by an equivalence of the form:

$$\forall(\mathcal{M}(Q) \Leftrightarrow G_1 \vee \ldots \vee G_k)$$

Each G_i is essentially the conjunction of the substitution θ_i and the meanings of the expressions in S_i in which all newly introduced variables are existentially quantified in front of G_i. However, what makes the result somewhat tedious to express is that skolemisation must be cancelled. G_i is of the form:

$$G_i = \exists \overline{X_i} : \theta_i' \wedge \sigma_i^{-1}(\mathcal{M}(F_1) \wedge \ldots \wedge \mathcal{M}(F_n)$$

Here θ_i' and σ_i are determined as follows. For all operators except the positive resolution and abduction operator, $\theta' = \sigma = \theta = \varepsilon$. For the abduction operator, $\sigma = \theta$ and $\theta' = \varepsilon$. The positive resolution operator produces a positive unifier $\theta = \sigma o \theta'$, where θ' is the result of the equality reduction and σ is the skolemising substitution.

Example In the previous example, the following equivalence is implicitly proven by SLDNFA:

$$\forall V : r(sk, V) \wedge q(f(sk, V)) \Leftrightarrow$$
$$sk = a \wedge V = b \wedge q(f(a, b)) \vee$$
$$\exists X, Z : sk = g(X) \wedge V = g(X) \wedge q(Z) \wedge q(f(g(X), g(X)))$$

Here, we have $\theta_1 = \theta'_1 = \{sk/a, V/b\}$ and $\sigma_1 = \varepsilon$ and $\theta_2 = \sigma_2 o \theta'_2 = \{X/sk_2\} o \{sk/g(X), V/g(X)\}$.

Proposition 5.4.3 (positive resolution) *Let Q' be a positive resolvent of Q and a variant of a program clause of P, using a positive unifier θ. It holds that:*

$$comp_3(\mathcal{L}, P) \models \forall(\theta(\&(Q)) \Leftarrow \&(Q'))$$

Let $\{Q_1, \ldots, Q_g\}$ be the set of positive resolvents of Q and a program clause, using the positive unifiers $\theta_1, \ldots, \theta_g$. Let G_i be the formula $\exists \overline{X}_i : \theta'_i \wedge \sigma_i^{-1}(\&(Q_i))$. Here $\theta_i = \sigma_i o \theta'_i$, with θ'_i the result of the equality reduction, σ_i the skolemising substitution and $\overline{X}_i = var(\theta'_i \wedge \sigma_i^{-1}(\&(Q_i))) \setminus var(Q)$. It holds that:

$$comp_3(\mathcal{L}, P) \models \forall(\&(Q) \Leftrightarrow (G_1 \vee \ldots \vee G_g))$$

Proof We prove the second item of the property. Let L_m be the selected atom $p(\overline{t})$. Take the suitable renaming of the definition of p/k in $comp_3(\mathcal{L}, P)$, corresponding to the renamings of the clauses in the definition of p/k used for positive resolution:

$$p(\overline{X}) \Leftrightarrow (\exists \overline{Y}_1 : \overline{X} = \overline{s}_1 \wedge B_1^1 \wedge \ldots, B_{q_1}^1) \vee \ldots \vee (\exists \overline{Y}_g : \overline{X} = \overline{s}_g \wedge B_1^g \wedge \ldots, B_{q_g}^g)$$

First, substitute $\overline{X}$ by $\overline{t}$ in this definition and replace L_m in $\&(Q)$ by the right hand of the definition.

Distribute the conjunction of $\&(Q)$ over the disjunction in the definition of p/k. Move the existential quantifiers of the definition of p/k outside the conjunction of $\&(Q)$. This is possible because these existentially quantified variables do not occur in the rest of the formula. After applying commutativity, we obtain a disjunction of formulas of the form $\exists \overline{Y} : \overline{t} = \overline{s} \wedge L_1 \wedge \ldots \wedge L_{m-1} \wedge B_1 \wedge \ldots \wedge B_q \wedge L_{m+1} \wedge \ldots \wedge L_k$.

There is a one to one mapping between the resolvents $Q_1, \ldots, Q_g$ and the disjuncts which contain a positively unifiable equality set $\overline{t} = \overline{s}_i$. Because of Proposition 5.4.1, we may remove all disjuncts for which the equality reduction fails and replace in the other $\overline{t} = \overline{s}_i$ by the corresponding solved form θ'. Finally apply θ' on the rest of the formula. We obtain exactly the desired formula.

The first item of the property can be proven in an analogous way, or can be derived directly from the second item: consider skolem constants as universal variables (Lemma 5.4.1); drop all but the disjunct corresponding to the

resolvent under consideration. This turns the $\Leftrightarrow$ equivalence into an $\Leftarrow$ implication. Move the existential quantifiers in the remaining disjunct to universal quantifiers in the front. Apply variable elimination on variables at the left in the substitution. This eliminates all equalities of the substitution. Finally skolemise all variables which correspond to skolem constants in Q and the resolvent. We obtain $\forall(\theta(\&(Q)) \Leftarrow \&(Q'))$.

□

Proposition 5.4.4 (abduction) *Let Q be a positive goal in which an abducible atom A is selected. Let Q' be obtained by abducing A using the skolemising substitution θ.*

The following formula is a tautology: $\forall(\theta(\&(Q)) \Leftarrow \theta(A) \wedge \&(Q'))$.

Let Q'' be the formula obtained by deleting A in Q. The following formula is a tautology: $\forall(\&(Q) \Leftrightarrow A \wedge \ \&(Q''))$.

Proof The proof is trivial. □

For negative resolution, two results are proven, one when applied to defined predicates, one when applied to abducible predicates.

Example Consider again the definition for $r/2$:

$r(a,b)$:-
$r(g(X),g(X))$:- $q(Z)$

and the negative goal $\leftarrow r(sk,V), q(f(sk,V))$. The negative resolution operator generates the negative goal $\leftarrow sk = a, q(f(sk,b))$ and the negative goal $\leftarrow sk = g(X), q(Z), q(f(sk,g(X)))$. We have the following equivalence:

$$\forall V : \neg r(sk,V) \vee \neg q(f(sk,V)) \Leftrightarrow (\neg sk = a \vee \neg q(f(sk,b))) \wedge (\forall X, Z : \neg sk = g(X) \vee \neg q(Z) \vee \neg q(f(sk,g(X))))$$

Proposition 5.4.5 (negative resolution) *Let $Q_1, \ldots, Q_g$ be the negative resolvents of Q and the clauses of P on L_m.*

$comp_3(\mathcal{L}, P) \models Q \Leftrightarrow Q_1 \wedge \ldots \wedge Q_g$

Let Q' be the negative resolvent of Q with a ground fact $A \leftarrow$ (which does not necessarily belong to P) : $\mathrm{FEQ}(\mathcal{L}) \models A \wedge Q \Leftrightarrow A \wedge Q \wedge Q'$

Proof We start with the first item. Assume L_m is $p(\overline{t})$ and p/k has definition:

$$\forall \overline{X} : p(\overline{X}) \Leftrightarrow (\exists \overline{Y}_1 : \overline{X} = \overline{s}_1 \wedge B_1^1 \wedge \ldots \wedge B_{q_1}^1) \vee \ldots \vee (\exists \overline{Y}_g : \overline{X} = \overline{s}_g \wedge B_1^g \wedge \ldots \wedge B_{q_g}^g)$$

This formula is equivalent with:

$$\forall \overline{X} : \neg p(\overline{X}) \Leftrightarrow \begin{array}{l}(\forall \overline{Y}_1 : \overline{\vee}(\leftarrow \overline{X} = \overline{s}_1, B_1^1, \ldots, B_{q_1}^1)) \wedge \ldots \wedge \\ (\forall \overline{Y}_g : \overline{\vee}(\leftarrow \overline{X} = \overline{s}_g, B_1^g, \ldots, B_{q_g}^g))\end{array}$$

We start with Q and rewrite it by applying equivalence preserving rules. Remember that Q stands for $\forall(\neg L_1 \vee \ldots \vee \neg L_m \vee \ldots \vee \neg L_k)$. First, substitute $\overline{X}$ by $\overline{t}$ everywhere in the negated definition and replace $\neg L_m$ in Q by the right hand of the negated definition. Distribute the universal quantifiers and the disjunction of Q over the conjunction of the negated definition of p/k. Move the universal quantifiers of the definition of p/k outside the disjunction of Q. This is possible because these universal variables are fresh. We obtain a conjunction of formulas of the form :

$$\leftarrow L_1, \ldots, L_{m-1}, \overline{t} = \overline{s}, B_1, \ldots, B_q, L_{m+1}, \ldots, L_k)$$

What remains to be done is to simplify this formula, according to the way negative unification works. By Proposition 5.4.1, we may replace $\overline{t} = \overline{s}$ by its equality reduction. If the unification of $\overline{t} = \overline{s}$ fails, $\overline{t} = \overline{s}$ is inconsistent and the conjunct is true and may be eliminated from the conjunction. Otherwise the negative unification of $\overline{t} = \overline{s}$ results in a residue E_r and a variable substitution θ. The application of θ on the negative resolvent is simulated by applying variable elimination on each universally quantified variable in the domain of θ.

To prove the second item, it suffices to show that:

$$FEQ(\mathcal{L}) \models A \wedge Q \Rightarrow Q'$$

	$p(\overline{s}) \wedge \leftarrow L_1, \ldots, p(\overline{t}), \ldots, L_k$	
$\Rightarrow$	$p(\overline{s}) \wedge \leftarrow L_1, \ldots, \overline{t} = \overline{s}, p(\overline{t}), \ldots, L_k$	(by subsumption)
$\Rightarrow$	$p(\overline{s}) \wedge \leftarrow L_1, \ldots, \theta, E_r, p(\overline{t}), \ldots, L_k$	(by equality reduction)
$\Rightarrow$	$p(\overline{s}) \wedge \leftarrow L_1, \ldots, \theta, E_r, p(\overline{s}), \ldots, L_k$	(by applying θ and E_r)
$\Rightarrow$	$p(\overline{s}) \wedge Q'$	(by elimination of θ and $p(\overline{s})$)
$\Rightarrow$	Q'	

□

Proposition 5.4.6 (switching to $\mathcal{NG}$ and $\mathcal{PG}$) *Let Q be a positive or negative goal in which a ground negative literal $\neg A$ is selected and Q' is obtained from Q by deleting $\neg A$. The following formula's are tautologies.*

$\forall(\&(Q) \Leftrightarrow \&(Q') \wedge \neg A)$	*(Switch to $\mathcal{NG}$)*
$Q \Leftrightarrow A \vee (\neg A \wedge Q')$	*(Switch to $\mathcal{PG}$ and Failed negation)*
$Q \Leftarrow A$	*(Switch to $\mathcal{PG}$)*
$Q \Leftarrow (\neg A \wedge Q')$	*(Failed negation)*

Proof The first item is trivial. The second item is only slightly less trivial. Since the literal A is ground, $\neg\neg A$ can be moved outside the universal quantifier. That $\neg A$ may be added to the second disjunct is an application of the tautology $p \vee q \Leftrightarrow p \vee (\neg p \wedge q)$. The third and fourth items are directly implied by the second. □

5.5 Soundness of SLDNFA

We will prove the following soundness result for the SLDNFA procedure with respect to completion semantics. Below, $P + \Delta$ denotes the complete program consisting of the incomplete program P and (possibly empty) definitions for the undefined predicates given by Δ.

Theorem 5.5.1 (soundness) *Let (Δ, θ) be the result of an SLDNFA refutation for a goal Q_0:*

$$<\mathcal{L} + Sk(\Delta), comp_3(\mathcal{L} + Sk(\Delta), P + \Delta)> \models \forall(\theta(\&(Q_0)))$$

Moreover, $<\mathcal{L} + Sk(\Delta), comp_3(\mathcal{L} + Sk(\Delta), P + \Delta)>$ is consistent.

The consistency of $comp_3(\mathcal{L} + Sk(\Delta), P + \Delta)$ follows easily from the fact that a logic program is always consistent and even always has a Herbrand model under each of the three justification semantics. Observe that the consistency would not be guaranteed under two-valued completion semantics. The following program in which r_1 and r_2 are abducible predicates illustrates the problem.

Example $P = \{p \text{ :- } \neg p, r_1 \quad q \text{ :- } r_1 \quad q \text{ :- } r_2\}$

SLDNFA generates two solutions for the goal $\leftarrow q$, namely $\{r_1\}$ and $\{r_2\}$. One easily observes that due to the first rule, $comp(P + \{r_1\})$ is inconsistent wrt two-valued semantics. However, the interpretation $\{p^{\mathbf{u}}, r_1^{\mathbf{t}}, q^{\mathbf{t}}\}$ is a model of $comp_3(P + \{r_1\})$ and of $comp_3(P)$. Under the constructive definition view (see chapter 4), the definition of p is locally inconsistent. P is not overall consistent.

The proof of the soundness result is in the spirit of Clark's soundness theorem for SLDNF [Cla78], but is seriously complicated by the skolem constants, by positive and negative abduction. The main idea behind the proof is that during the construction of an SLDNFA-derivation K of length n, a *proof tree* is constructed. Initially, the root and only node of the proof tree is labelled by $\mathcal{M}(Q_0)$, the meaning of the query. The selection of a goal Q corresponds to the selection of a leaf of the proof tree with label $\mathcal{M}(Q)$. An operation which produces a substitution θ and a set $\{F_1, \ldots, F_k\}$ of expressions, applies θ on all existing nodes and extends the proof tree at the node labelled $\mathcal{M}(Q)$ with descendants labelled

$\mathcal{M}(F_1), \ldots, \mathcal{M}(F_k)$. This way, we obtain a tree with root $\theta_n o \ldots o \theta_1(\mathcal{M}(Q_0))$ and leaves corresponding to the elements of $\mathcal{PG}_n, \mathcal{NG}_n$ and Δ_n. The case for the negative abductive operator is slightly different. Note that this operator deletes a tuple (Q, A_Q, D_Q) from $\mathcal{NAG}$ and adds a negative goal Q' and a tuple with the same negative abductive goal Q. The proof tree contains a node labelled Q. Instead of adding Q' and Q as descendants to this node, only Q' is added.

In addition, the tree contains for each node some status word, which is a string of the set $\{pg, ng, nag, abd, comp\}$. The status of a node labelled by $\mathcal{M}(Q)$ indicates to what (multi)set Q belongs:

- *pg*: $Q \in \mathcal{PG}_n$
- *ng*: $Q \in \mathcal{NG}_n$
- *nag*: $(Q, A_Q, D_Q) \in \mathcal{NAG}_n$
- *abd*: $Q \in \Delta_n$
- *comp*: Q is a goal to which some inference step has already been applied.

We give an inductive definition of the proof tree.

Definition 5.5.1 *Let K be a finite SLDNFA derivation of length n for a goal Q_0.*

If $n = 0$, we define the proof tree $\mathcal{P}_K$ associated to K as the tree with a single node, labelled with the open formula $\&(Q_0)$ and status pg.

Assume $n > 0$. Let K' be the SLDNFA derivation consisting of the first $n-1$ steps of K. An operation is applied on a selection (Q, L_m) or $((Q, A_Q, D_Q), B)$ and produces a substitution θ and a set S of expressions. The proof tree $\mathcal{P}_K$ of K is constructed from $\mathcal{P}_{K'}$ as follows:

- *let the selection be of the form (Q, L_m) with $Q \in \mathcal{PG}_{n-1}$ or $Q \in \mathcal{NG}_{n-1}$. Let $N \in \mathcal{P}_{K'}$ be the corresponding leaf labelled $\mathcal{M}(Q)$. $\mathcal{P}_K$ is obtained from $\mathcal{P}_{K'}$ by applying θ on all labels of $\mathcal{P}_{K'}$ and appending to N for each $F \in S$ one new descendant N', labelled with $\mathcal{M}(F)$ and the status of F. The status of N is changed to comp.*
- *let the selection be $((Q, A_Q, D_Q), B)$. The $\mathcal{NAG}$ operator produces a negative resolvent Q' and the tuple (Q, A_Q, D'_Q). Let $N \in \mathcal{P}_{K'}$ be the node corresponding to (Q, A_Q, D_Q) and labelled by Q. $\mathcal{P}_K$ is obtained from $\mathcal{P}_{K'}$ by appending to N a new descendant N' labelled with $\mathcal{M}(Q')$ and status ng.*

Example The proof tree of the first SLDNFA refutation in the faulty lamp example (section 5.3) for the goal $\leftarrow$ *faulty_lamp* is given in figure 5.1.

The proof of the soundness goes as follows. For each node N with status *comp*, a goal Q was selected at step i of the derivation and N is labelled by

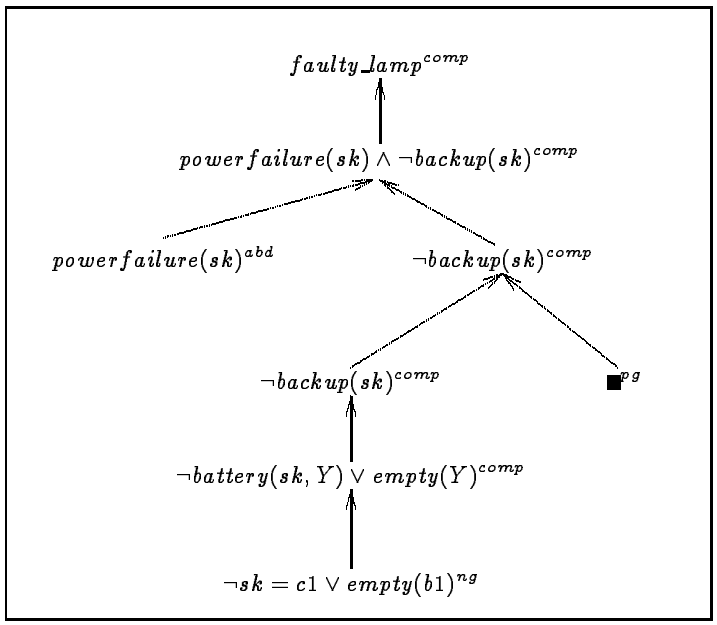

Figure 5.1: Proof tree of $\leftarrow faulty\_lamp$

$\theta_n o \ldots \theta_i(\mathcal{M}(Q))$. The descendants of N are labelled by $\theta_n o \ldots o\theta_{i+1}(\mathcal{M}(F_j))$ for each F_j, produced by the i'th operation. It follows directly from the theorems of subsection 5.4.2 that each formula in a node with status *comp* is implied by the conjunction of the formulas in the descendants.

The proof of the soundness proceeds by showing that for a refutation, all leaves of the corresponding final proof tree hold in $comp_3(\mathcal{L} + Sk(\Delta), P + \Delta)$ and that nodes with status *nag* are implied by the conjunction their descendants. By a simple induction, it is then possible to derive the correctness of the root $\theta_a(\&(Q_0))$ with θ_a the answer substitution of the refutation[4].

At this point, we can clarify why $\mathcal{PG}$, $\mathcal{NG}$ and $\mathcal{NAG}$ are designed as multisets and not as simple sets. If these were sets, then it can occur that a goal added by

[4] An alternative proof would be to show that with any SLDNFA derivation, one can associate an SLDNF refutation in $P + \Delta$ generating the same answer substitution, and then relying on the soundness of SLDNF. Though at first sight this may seem easier, there are definitely a number of tedious snakes under the grass (see for example [MBD92]). Moreover this technique is also substantially weaker. Using the proof tree, we obtain the additional result that the conjunction of the leaves of the proof tree (the conjunction consisting of $\&(\Delta)$ and all remaining constraints) imply $\theta(\&(Q))$ under the completion of the incomplete program P. This is an interesting result in its own right and it will prove useful later.

some computation step is identical to an existing goal, so they *merge*. Alternatively, due to the application of a substitution, two previously different goals may become identical and merge. The danger of this is that -if no care is taken- this might lead to a proof tree which contains loops. Indeed, examples can be constructed in which a negative abductive goal (Q, A_Q, D_Q) has an identical ancestor goal. If in the proof tree, the nodes for these goals are merged then we obtain a proof tree with a loop.

We prove that, given a refutation K, all the leaves of $\mathcal{P}_K$ are implied by the completion of the program augmented with the abductive solutions and that each node is implied by its descendants.

Lemma 5.5.1 *Let K be an SLDNFA-refutation of length n, generating a solution (Δ, θ). For every node N of $\mathcal{P}_K$, with label F and descendants labelled with $F_1, \ldots, F_m$ ($m \geq 0$), it holds that $comp_3(\mathcal{L} + Sk(\Delta), P + \Delta) \models \forall(F \Leftarrow F_1 \wedge \ldots \wedge F_m)$.*

The case that $m = 0$ corresponds to a leaf of the proof tree. In that case, the empty conjunction corresponds to true*, so that the truth of the leaf must be proven.*

Proof Depending on the status of the node, the tree is partitioned into the 5 classes of nodes $\mathcal{N}_{comp}$, $\mathcal{N}_{abd}$, $\mathcal{N}_{pg}$, $\mathcal{N}_{ng}$ and $\mathcal{N}_{nag}$. The proof is by a case analysis on the type of the node.

- $N \in \mathcal{N}_{abd}$: An abductive node is always a leaf and is labelled with an abducible atom contained in Δ.
- $N \in \mathcal{N}_{pg}$: N is labelled with $\mathcal{M}(\Box)$, this is the empty conjunction.
- $N \in \mathcal{N}_{ng}$: N is labelled with the meaning of a negative goal Q containing an irreducible equality atom $sk = t$. The result follows immediately from $FEQ(\mathcal{L} + Sk(K)) \models \forall(\neg sk = t)$.
- $N \in \mathcal{N}_{nag}$: N corresponds to a negative abductive goal (Q, A_Q, D_Q). N is labelled with $\mathcal{M}(Q) = Q$. Assume that the descendants of N are labelled with formulas $F_1, \ldots, F_h$. It suffices to show that $comp_3(\mathcal{L}, \Delta) \models Q \Leftarrow F_1 \wedge \ldots \wedge F_h$.

  We know that D_Q contains every atom from Δ unifiable with the selected atom A_Q. Let $Q_1, \ldots, Q_g$ be the negative resolvents of Q and an atom of D_Q. From Proposition 5.4.5, it follows that $comp_3(\mathcal{L}, \Delta) \models Q \Leftrightarrow Q_1 \wedge \ldots \wedge Q_g$. What remains to be proven is that for each Q_i, there exists an F_j such that $F_j \Leftrightarrow Q_i$.

  Take any Q_i. It is obtained by negative resolution with an abduced fact $B = p(\overline{s})$ of D_Q. Therefore, there exists at least one descendant N_j of N, obtained at some step l by negative resolution of a negative abductive goal Q' with an abduced fact $B' = p(\overline{s}')$ such that $\theta_n o \ldots o \theta_l(Q')$ is Q and $\theta_n o \ldots o \theta_l(B')$ is B. Let Q_i' be the result of this negative resolution. The label F_j of N_j is nothing else than $\theta_n o \ldots o \theta_l(Q_i')$. We must show that F_j is equivalent with Q_i.

Let $p(\overline{t}')$ be the selected goal in Q' and $p(\overline{s}')$ be the atom B'. Obviously, $\theta_n o \ldots o \theta_l(p(\overline{t}'))$ is $A_Q = p(\overline{t})$. Consider the goal G' obtained from Q' by replacing $p(\overline{t}')$ by $\overline{t}' = \overline{s}'$. From the soundness of equality reduction (Proposition 5.4.1), it follows that $FEQ(\mathcal{L}) \models G' \Leftrightarrow Q_i'$. This equivalence is preserved under the application of the substitution $\theta_n o \ldots o \theta_l$. Under this substitution, Q_i' becomes F_j and G' becomes the goal G obtained by replacing A_Q in Q by $\overline{t} = \overline{s}$. Again by the correctness of equality reduction, it follows that $FEQ(\mathcal{L}) \models G \Leftrightarrow Q_i$. We obtain at last that $FEQ(\mathcal{L}) \models F_j \Leftrightarrow Q_i$.

$N \in \mathcal{N}_{comp}$: With N corresponds a goal Q that was selected at step i such that N is labelled by $\theta_n o \ldots o \theta_i(\mathcal{M}(Q)))$. The descendants of N are labelled by $\theta_n o \ldots o \theta_{i+1}(\mathcal{M}(F_j))$ for each F_j that was produced by the i'th operation. To prove is $\forall(\theta_n o \ldots o \theta_{i+1}(\theta_i(\mathcal{M}(Q)) \Leftarrow \mathcal{M}(F_1) \wedge \ldots \wedge \mathcal{M}(F_h)))$. This is a direct consequence of the propositions of subsection 5.4.2) and the fact that the application of a substitution on universally quantified variables and skolem constants preserves a logical consequence.

□

Proof (of theorem 5.5.1)

Let K be an SLDNFA-refutation for the goal Q_0, with proof tree $\mathcal{P}_K$. In Lemma 5.5.1, we have proven that for each node labelled with a formula F, with descendants labelled with $F_1, \ldots, F_m$, it holds that:

$$comp_3(\mathcal{L} + Sk(\Delta), P + \Delta) \models \forall(F \Leftarrow F_1 \wedge \ldots \wedge F_m)$$

Using a simple induction on the depth of the nodes in the proof tree, one obtains that for each formula F in any node of the tree: $comp_3(\mathcal{L} + Sk(\Delta), P + \Delta) \models \forall(F)$. This holds a fortiori for the formula in the root of the tree, which is $\forall(\theta(\&(Q_0)))$.

Recall that the underlying language is $\mathcal{L}_{sk}$. Since $comp_3(\mathcal{L} + Sk(\Delta), P + \Delta)$ contains only symbols of $\mathcal{L} + Sk(\Delta)$, we may restrict our language to the latter language and we obtain:

$$<(\mathcal{L} + Sk(\Delta), comp_3(\mathcal{L} + Sk(\Delta), P + \Delta)> \models \forall(F)$$

That $comp_3(\mathcal{L} + Sk(\Delta), P + \Delta)$ is consistent, follows from the duality theorem 4.6.1 and theorem 4.4.1 which proves the equivalence between 3-valued completion semantics and direct justification semantics. □

5.6 Completeness of SLDNFA

5.6.1 The completeness theorems

To formulate the completeness result, the concept of SLDNFA-tree is needed. First we define how the computation *branches*: given some SLDNFA-derivation and a selection in the last node of the derivation, what SLDNFA operations are possible on this selection? We then define the concept of an SLDNFA-tree as a closure over the branching relation.

Definition 5.6.1 *Given is a finite SLDNFA derivation K of length $n-1$ and a selection (Q, L) or $((Q, A_Q, D_Q), B)$ at step $n-1$.*

The set S of computable children of K is the set of all tuples $(\mathcal{PG}_n^i, \mathcal{NG}_n^i, \mathcal{NAG}_n^i, \Delta_n^i, \theta_n^i)$, that can be obtained by applying one of the SLDNFA-operations on the selected tuple.

In practice, only in two situations more than one child can exist:

- when Q is selected from $\mathcal{PG}_{n-1}$ and L_m is a non-abducible atom, then there are as many children as there are program clauses with a head positively unifiable with L_m.
- when Q is selected from $\mathcal{NG}_{n-1}$ and L_m is a negative literal, then the switch to $\mathcal{PG}$ and the failed negation operations can be applied and we obtain two children.

Definition 5.6.2 *A partial SLDNFA-tree for a query Q is a tree in which each branch is an SLDNFA derivation. For each non-leaf N, a selection exists such that the descendants of N are the computable children of the derivation up to N.*

An SLDNFA-tree for a query Q is a partial SLDNFA-tree such that each branch is a failed SLDNFA derivation or an SLDNFA refutation.

Example The SLDNFA-tree for the query $\leftarrow$ *faulty_lamp* in the faulty lamp example is given in figure 5.2.

Observe that the SLDNFA-tree is a totally different concept than the proof tree! The proof tree represents one SLDNFA-derivation in a structured way, while an SLDNFA-tree represents a set of SLDNFA-derivations. So with every branch of an SLDNFA-tree, a proof tree corresponds.

As a completeness result, we will prove that (under certain conditions expressed below), SLDNFA generates all minimal and most general solutions. More precisely: given any abductive solution Δ, there exists a solution Δ' generated by SLDNFA and a substitution σ for the skolem constants of Δ' such that $\sigma(\Delta') \subseteq \Delta$. This does not imply that Δ' contains less elements than Δ: indeed it is possible that σ maps two or more facts of Δ' to one fact of Δ. In the section on extensions of SLDNFA we come back to this phenomenon.

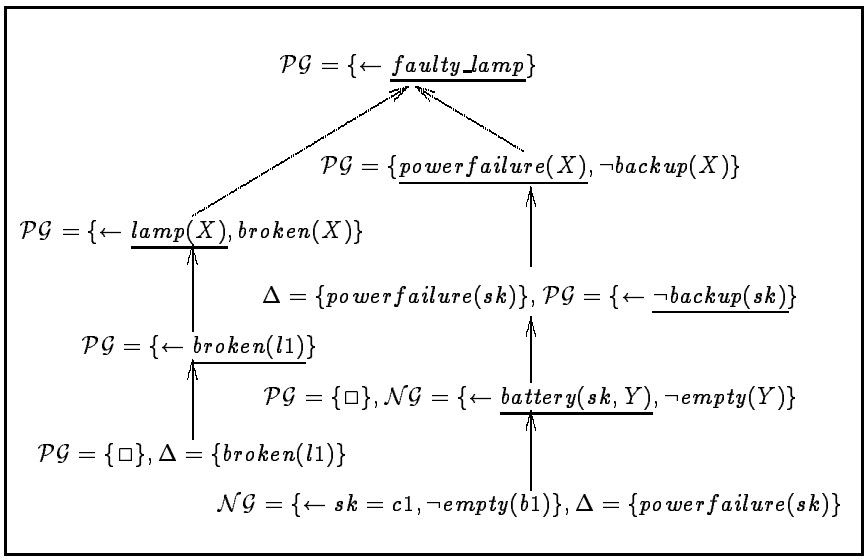

Figure 5.2: SLDNFA-tree of $\leftarrow faulty\_lamp$

Theorem 5.6.1 (completeness) *Given is a normal abductive program P based on a language $\mathcal{L}$ and a normal query Q which has a finite SLDNFA-tree W.*

(a) *if all branches of W are finitely failed, then $<\mathcal{L}, comp_3(\mathcal{L}, P)> \models Q$*

(b) *if $<\mathcal{L}, comp_3(\mathcal{L}, P) + \exists(\&(Q))>$ is satisfiable, then W contains a successful branch.*

(c) *let (Δ, θ) be an abductive solution for Q based on an extension $\mathcal{L}'$ of $\mathcal{L}$. There exists a successful branch in W generating a solution (Δ', θ') and a ground skolem substitution σ_{sk} such that $\sigma_{sk}(\Delta') \subseteq \Delta$. Moreover $\sigma_{sk} o \theta'(Q)$ is more general than $\theta(Q)$.*

An important observation is that due to item (a), SLDNFA can be used in a totally different role than for abduction: it can be used soundly for deduction in incomplete logic programs. Indeed, assume that we want to prove a formula F wrt an incomplete program P (under completion semantic or, (direct) (partial) justification semantics). Consider the general query $\leftarrow \neg F$. By applying the Lloyd-Topor transformation [LT84] (see algorithm 4.9.1), this general query can be transformed into normal query $Q = \leftarrow L_1, \ldots, L_n$ and a program P'. Moreover by theorem 4.9.1, it holds that $P \models F$ iff $P + P' \models \leftarrow L_1, \ldots, L_n$ under completion semantics. If SLDNFA fails on Q, then by theorem 5.6.1(a) we find that

$comp_3(P) \models F$. By theorem 4.3.3, this implies that F is entailed by P under (partial) justification semantics. This is again a proof of the remarkable versatility of abduction. This application of SLDNFA will be illustrated in chapter 6. Note that SLDNFA is not complete as a theorem prover, due to the problems with floundering.

SLDNFA, like SLDNF, can only be complete under severe restrictions. The condition of having a finite computation tree is quite restrictive. It is needed because the proof relies on an equivalence between the goal and the leafs of the SLDNFA-tree. This equivalence can only be proven for finite trees. This proof technique is in the same spirit as Clark's proof of the completeness of SLDNF for hierarchical programs. Similarly as for SLDNF, completeness results for infinite SLDNFA-trees could probably be obtained by imposing conditions such as allowedness and strictness on the abductive programs (see [CL89]). This is subject to future research.

The proof of the completeness theorems is also in the spirit of Clark's completeness proof for SLDNF. We show that with a finite SLDNFA-tree, a formula can be associated which expresses the equivalence between the goal and a disjunction of formulas each of which corresponds to one successful refutation in the SLDNFA-tree. Following [CTT91], we call this formula the *explanation formula* and the disjuncts corresponding to the SLDNFA refutations the *state formulas*. In the following subsection, the explanation formula is introduced and its correctness is proven. After that, the explanation formula is used to prove the completeness.

5.6.2 The explanation formula

Below $\mathcal{M}_{sk}(F)$ denotes $\mathcal{D}^{-1}(\mathcal{M}(F))$, the deskolemisation of the meaning of F. For a set S of expressions, we define $\mathcal{M}_{sk}(S)$ as the conjunction of the deskolemised meanings of its elements. For a substitution θ, we define $\mathcal{M}_{sk}(\theta)$ as the conjunction of the equalities obtained by deskolemising all equalities in θ and removing all atoms of the form $X = X$. Such an atom is the result of deskolemising an equality $X = \mathcal{D}(X)$.

Definition 5.6.3 *Let P be a normal abductive program, Q_0 a query, K a finite SLDNFA derivation of length n, θ_a the answer substitution $\theta_n o \ldots o \theta_1|_{var(Q_0)}$*

The state formula corresponding to K is the open formula:

$$\exists \overline{Y} : E \wedge F_{abd} \wedge F_{pg} \wedge F_{ng}$$

where E is $\mathcal{M}_{sk}(\theta_a)$, the deskolemised answer substitution; F_{abd} is $\mathcal{M}_{sk}(\Delta)$, the deskolemisation of Δ; F_{pg} is $\mathcal{M}_{sk}(\mathcal{PG}_n)$, the conjunction of the deskolemised meanings of the positive goals; F_{ng} is $\mathcal{M}_{sk}(\mathcal{NG}_n \cup \mathcal{NAG}_n)$, the conjunction of the deskolemised meanings of negative and negative abductive goals. $\overline{Y}$ is $var(E \wedge F_{abd} \wedge F_{pg} \wedge F_{ng}) \setminus var(Q_0)$. The state formula is denoted by $state(K)$.

Definition 5.6.4 *Given a finite partial SLDNFA-tree W for a goal Q_0, its explanation formula is:*

$$\forall(\&(Q_0) \Leftrightarrow state(K1) \vee \ldots \vee state(K_g))$$

where $\{K_1, \ldots, K_g\}$ is the set of all branches of W which are not failed. The formula is denoted by $expl(W)$.

Example Consider the SLDNFA-tree of figure 5.2. The corresponding explanation formula is:

$$\begin{aligned} faulty\_lamp \Leftrightarrow\ & broken(l1) \vee \\ & (\exists X : powerfailure(X) \wedge (\neg X = c1 \vee \neg\neg empty(b1))) \end{aligned}$$

Observe that the explanation formula does not contain skolems. Therefore, $\mathcal{L}$ is taken as the underlying language. With respect to the completeness, we will first show that for every finite partial SLDNFA-tree, $comp_3(\mathcal{L}, P) \models expl(W)$. This theorem is based on the following lemma.

Lemma 5.6.1 *For any SLDNFA-derivation K of length n, let there be a selection in the last node of K and $\{K_1, \ldots, K_g\}$ the set of extensions of K with a computable child of K. Then the following equivalence holds:*

$$comp_3(\mathcal{L}, P) \models \forall(state(K) \Leftrightarrow state(K_1) \vee \ldots \vee state(K_g))$$

Proof The proof of the lemma is a straightforward extension of the completeness results proven in subsection 5.4.2. K is extended to $K_1, \ldots, K_g$ by performing all possible operations on a selection (Q, L_m) or $((Q, A_Q, D_Q), B)$. Let $S_1, \ldots, S_g$ be the produced sets of expressions by each operation, $\theta_1, \ldots, \theta_g$ the produced substitutions. As in subsection 5.4.2, each θ_i can be written as $\sigma_i o \theta_i'$ with σ_i the skolemising part of θ_i. By the theorems of subsection 5.4.2, it holds that:

$$comp_3(\mathcal{L}, P) \models \forall(\mathcal{M}(Q) \Leftrightarrow G_1 \vee \ldots \vee G_g)$$

G_i corresponds to $S_i = \{F_1, \ldots, F_h\}$ as follows:

$$G_i = \exists \overline{X}_i : \theta_i' \wedge \sigma_i^{-1}(\mathcal{M}(F_1) \wedge \ldots \wedge \mathcal{M}(F_h))$$

where $\overline{X}_i$ is the set of variables appearing in $\theta_i' \wedge \sigma_i^{-1}(\mathcal{M}(F_1) \wedge \ldots \wedge \mathcal{M}(F_h))$ but not in $\mathcal{M}(Q)$. By lemma 5.4.1, the following equivalence holds:

$$\forall(\mathcal{M}_{sk}(Q) \Leftrightarrow \mathcal{D}^{-1}(G_1) \vee \ldots \vee \mathcal{D}^{-1}(G_g))$$

and $\mathcal{D}^{-1}(G_i)$ is equivalent with:

$$\exists \overline{X}_i : \mathcal{M}_{sk}(\theta_i) \wedge \mathcal{M}_{sk}(F_1) \wedge \ldots \wedge \mathcal{M}_{sk}(F_h)$$

The proof goes as follows. $state(K)$ is of the form $\exists\overline{Y} : E \wedge F_{abd} \wedge F_{pg} \wedge F_{ng}$ and contains $\mathcal{M}_{sk}(Q)$. Substituting $\mathcal{D}^{-1}(G_1) \vee \ldots \vee \mathcal{D}^{-1}(G_g)$ for $\mathcal{M}_{sk}(Q)$ preserves the equivalence. The next step is to simplify the resulting formula to obtain $state(K_1) \vee \ldots \vee state(K_g))$. This is easy when the abduction operator, switch to $\mathcal{NG}$, negative resolution operator, move to $\mathcal{NAG}$ or the $\mathcal{NAG}$ operator is applied. In that case, only one computable child K_1 exists and moreover, $\theta_1' = \varepsilon$ and $\overline{X}_1 = \phi$. We immediately obtain $state(K_1)$.

In case that a negative literal is selected in a negative goal, two children K_1 and K_2 are obtained by applying the switch to $\mathcal{PG}$ and the failed negation operator. $\theta_1', \theta_2', \sigma_1, \sigma_2$ and $\overline{X}_1, \overline{X}_2$ are empty. It suffices to distribute the conjunction and existential quantifiers of $state(K)$ over the disjunction of $\mathcal{D}^{-1}(G_1) \vee \mathcal{D}^{-1}(G_2)$ to obtain $state(K_1) \vee state(K_2)$.

Finally, consider the case that $Q \in \mathcal{PG}_n$ and L_m is a defined atom. K has as many extensions $K_1, \ldots, K_g$ as there are positive resolvents $Q_1, \ldots, Q_g$. The first step of the proof is again to substitute $\mathcal{D}^{-1}(G_1) \vee \ldots \vee \mathcal{D}^{-1}(G_g)$ for $\mathcal{M}_{sk}(Q)$ in F_{pg} and to distribute the conjunction and existential quantifiers of $state(K)$ over this disjunction. We obtain a disjunction in which each disjunct is of the form: $\exists\overline{Y} : E \wedge F_{abd} \wedge F'_{pg} \wedge F_{ng} \wedge \exists\overline{X}_i : \mathcal{M}_{sk}(\theta_i) \wedge \mathcal{M}_{sk}(Q_i)$. Here F'_{pg} denotes the conjunction obtained by eliminating $\mathcal{M}_{sk}(Q)$ from F_{pg}. Since all variables of $\overline{X}_i$ are fresh, their existential quantifiers can be moved outside the conjunction, joining the quantifiers of $\overline{Y}$. We obtain $\exists\overline{Y}, \overline{X}_i : E \wedge \mathcal{M}_{sk}(\theta_i) \wedge F_{abd} \wedge F'_{pg} \wedge \mathcal{M}_{sk}(Q_i) \wedge F_{ng}$.

The next step is to apply $\mathcal{M}_{sk}(\theta_i)$ on $E \wedge F_{abd} \wedge F'_{pg} \wedge F_{ng}$. Since $\mathcal{M}_{sk}(\theta_i)$ does not contain variables of the domain of E, this transforms $E \wedge \mathcal{M}_{sk}(\theta_i)$ to $\mathcal{M}_{sk}(\theta_i) o E$. Also, $F_{abd} \wedge F'_{pg} \wedge \mathcal{M}_{sk}(Q_i) \wedge F_{ng}$ is transformed in $F^i_{abd} \wedge F^i_{pg} \wedge F^i_{ng}$.

The remainder of the transformation is to simplify $\mathcal{M}_{sk}(\theta_i) o E$ to the answer substitution of K_i. Let θ_a be the answer substitution of K, it holds that $E = \mathcal{M}_{sk}(\theta_a)$. θ^i_a is $\theta_i o \theta_a|_{var(Q)}$. One easily verifies that $\mathcal{M}_{sk}$ and the composition operator o commute. Therefore $\mathcal{M}_{sk}(\theta_i o \theta_a)$ is $\mathcal{M}_{sk}(\theta_i) o E$. The restriction of $\mathcal{M}_{sk}(\theta_i) o E$ to $var(Q)$ is obtained by applying variable elimination on all existentially quantified variables which occur in the domain of the resulting substitution. We obtain $state(K_i)$.

□

In the following theorem, the lemma is extended through an inductive argument.

Theorem 5.6.2 *For any finitely failed derivation K, $state(K)$ is inconsistent. For any finite SLDNFA-tree W, $comp_3(\mathcal{L}, P) \models expl(W)$*

Proof For the first item, observe that a derivation of length N is finitely failed iff $\mathcal{NG}_n$ contains the empty goal $\Box$ or if some positive goal has a positive literal L_m which is not unifiable with the head of any program clause. In the first case the inconsistency is trivial. In the second case: since the set of children of the derivation is empty, it follows from Lemma 5.6.1 that $state(K) \Leftrightarrow \Box$.

The second item follows by induction on the depth of the SLDNFA-tree, using Lemma 5.6.1.

□

5.6.3 Completeness proof

Proof (of theorem 5.6.1)

(a) When W is finitely failed, $expl(W)$ is of the form $\forall(\&(Q) \Leftrightarrow false)$ or equivalently Q. By theorem 5.6.2, $comp_3(\mathcal{L}, P) \models Q$.

(b) follows directly from (a).

(c) We assume that new variables introduced by θ do not occur in the SLDNFA-tree. Also we assume that $\mathcal{L}' \cap \mathcal{SK} = \phi$. These conditions can always made to hold, by renaming some variables or constants. By assumption $<\mathcal{L}', comp_3(\mathcal{L}', P + \Delta)> \models \forall(\theta(\&(Q)))$. Observe that $FEQ(\mathcal{L}')$ contains disequality constraints for all constants in $\mathcal{L}'$. This implies that the extra constants in $\mathcal{L}$' do not behave as skolem constants, e.g. they cannot occur at the left in an mgu.

Intuitively, the proof goes as follows. Take a model M of $comp_3(\mathcal{L}', P + \Delta)$. Since $comp_3(\mathcal{L}, P)$ is comprised in $comp_3(\mathcal{L}', P + \Delta)$, by theorem 5.6.2, the explanation formula is satisfied in M. Therefore for an instance of the variables of Q which satisfies $\&(Q)$, there must exists a $state(K_i)$ which is also satisfied under this variable assignment. $state(K_i)$ is of the form $\exists \overline{Y} : E \wedge F_{abd} \wedge F_{ng}$, so the variable assignment can be extended to the variables $\overline{Y}$, such that $E \wedge F_{abd} \wedge F_{ng}$ holds. In particular F_{abd} is satisfied. Since F_{abd} is satisfied, its conjuncts must correspond to abduced atoms of Δ. On the other hand, F_{abd} is the deskolemisation of some generated solution Δ'. Its variables correspond to skolem constants in Δ'. So we get a relation between terms in Δ' and in Δ. From this relation, the desired substitution can be obtained. Below, a precise formulation of this reasoning is given.

We construct the model M as follows. Take the Herbrand interpretation of a language $\mathcal{L}' + \{c_1, \ldots, c_n\}$, where $c_1, \ldots, c_n$ are new constants not appearing in $\mathcal{L}'$ and n is the number of variables in $\theta(Q)$. Extend this to an incomplete Herbrand interpretation by interpreting "=" by $\{t = t^{\mathbf{t}} \mid t \in HU(\mathcal{L}' + \{c_1, \ldots, c_n\})\}$. By theorem 4.4.1 and theorem 4.3.4, this incomplete interpretation can be extended to a model M of $comp_3(\mathcal{L}', P + \Delta)$.

In M, the formula $\forall(\theta(\&(Q)))$ holds. Take the special variable assignment V which associates to each distinct variable X_i of $\theta(Q)$ the new constant c_i. It holds that $M \models V(\theta(\&(Q)))$. This is equivalent to $M \models V(\exists Z_1, \ldots, Z_n : (\theta \wedge \&(Q)))$ where $Z_1, \ldots, Z_n$ are the variables of the domain of θ. Thus, there exists an extension V' of V such that $M \models V'(\theta \wedge \&(Q))$.

M satisfies $expl(W)$. Since $M \models V'(\&(Q))$, it must hold that $M \models V'(state(K_1) \vee \ldots \vee state(K_n))$[5]. Therefore, there exists an i such that $M \models V'(state(K_i))$. $state(K_i)$ is a formula of the form $\exists \overline{Y} : \theta'_{desk} \wedge F_{abd} \wedge F_{ng}$, where F_{abd} is the deskolemisation of the generated Δ' and θ'_{desk} is the deskolemisation of the generated answer substitution θ'. Again V' can be extended to V'' such that $M \models V''(\theta'_{desk} \wedge F_{abd} \wedge F_{ng})$.

Since $M \models V''(\theta)$ and $M \models V''(\theta'_{desk})$, we have for each term t^Q occurring in Q, that $M \models V''(t^Q = \theta(t^Q))$ and $M \models V''(t^Q = \theta'_{desk}(t^Q))$. By transitivity of "=", $M \models V''(\theta(t^Q) = \theta'_{desk}(t^Q))$.

Also $M \models V''(F_{abd})$. F_{abd} is a conjunction of abducible atoms $A_1 \wedge \ldots \wedge A_m$. Let each A_j be of the form $p_j(\overline{t}_j)$ and let p_j be defined by: $\forall \overline{X}$: $p_j(\overline{X}) \Leftrightarrow \overline{X} = \overline{s}^j_1 \vee \ldots \vee \overline{X} = \overline{s}^j_{n_j}$ in $comp_3(\mathcal{L}', P + \Delta)$. Keep in mind that if $p_j = p_i$, then these definitions are identical. From the fact that M is a model of this definition and that $M \models V''(A_j)$, apparently for each j, there must exist a i_j such that $M \models V''(\overline{t}_j = \overline{s}^j_{i_j})$.

Now summarizing, we have derived that under M and V'' the following conjunction C is satisfied : $\theta(t^Q_1) = \theta'_{desk}(t^Q_1) \wedge \ldots \wedge \theta(t^Q_l) = \theta'_{desk}(t^Q_l) \wedge \overline{t}_1 = \overline{s}^1_{i_1} \wedge \ldots \wedge \overline{t}_m = \overline{s}^m_{i_m}$, where $t^Q_1, \ldots, t^Q_l$ are all the terms of Q. Therefore, $\exists(C)$ is satisfiable with respect to $FEQ(\mathcal{L}')$. Now we can apply the reduction theorem (Proposition 5.4.1) which says that equality reduction yields a consistent solved form σ of C. σ is an mgu of C and is based on $\mathcal{L}'$.

By Proposition 5.4.1, $FEQ(\mathcal{L}') \models \forall(C \Leftrightarrow \sigma)$. As a result, $M \models V''(\sigma)$ [6]. This fact allows us to rename σ as follows. Let $\{X_1 = t_1, \ldots, X_l = t_l\}$ be the equalities of σ such that $X_i \in var(\theta(Q))$. It holds that $M \models V''(X_i = t_i)$. Since $V''(X_i) = V(X_i) = c_i$ is a constant which does not occur in $\mathcal{L}'$, t_i must be a variable Y_i and Y_i cannot be another variable of $\theta(Q)$ since V'' assigns distinct constants c_i to distinct variables of $\theta(Q)$. We define σ' as $\{Y_1 = X_1, \ldots, Y_l = X_l\} o \sigma$. It is clear that σ' is still an mgu of C. Moreover, $dom(\sigma') \cap var(\theta(Q)) = \phi$ and hence $\sigma'(\theta(Q)) \equiv \theta(Q)$.

We have that $\sigma'(\overline{t}_j) \equiv \sigma'(\overline{s}^j_{i_j})$. Therefore $\sigma'(A_j) \equiv \sigma'(p_j(\overline{s}^j_{i_j})) \in \sigma'(\Delta)$. Each atom $p_j(\overline{s}^j_{i_j})$ in Δ is ground and is based on $\mathcal{L}$'. Therefore $\sigma'(\Delta) \equiv \Delta$ and

[5] Observe that here we use the fact that the set of free variables of $state(K_1)) \vee \ldots \vee state(K_n)$ is a subset of $var(\&(Q))$. Otherwise, there would be need to extend V' to the additional variables.

[6] Again, we implicitly use the fact the variables of σ are a subset of the variables of C. This follows from the fact that equality reduction does not introduce new variables. Otherwise, there could be need to extend V'' to these additional variables.

we find that $\sigma'(F_{abd}) \subseteq \Delta$. Since all facts in Δ are ground, $var(F_{abd}) \subseteq domain(\sigma')$ and σ' is ground on all variables in F_{abd}. Now define $\sigma_{sk} = \{\mathcal{D}(X) = t \mid X = t \in \sigma' \text{ and } X \in var(F_{abd})\}$ (with $\mathcal{D}$ the skolemisation mapping). By construction, $\sigma_{sk}(\Delta') \equiv \sigma(F_{abd}) \subseteq \Delta$.

We have also that for each term t^Q in Q : $\sigma'(\theta'_{desk}(t^Q)) \equiv \sigma'(\theta(t^Q))$. Hence, $\sigma'(\theta'_{desk}(Q)) \equiv \sigma'(\theta(Q)) \equiv \theta(Q)$. By the way σ_{sk} is constructed out of σ' it follows straightforwardly that $\sigma' o \sigma_{sk}(\theta'(Q)) \equiv \sigma'(\theta'_{desk}(Q))$. The following equalities hold: $\sigma' o \sigma_{sk}(\theta'(Q)) \equiv \sigma'(\theta'_{desk}(Q)) \equiv \theta(Q)$. Thus $\sigma_{sk}(\theta'(Q))$ is more general than $\theta(Q)$. This concludes the proof of item (c). □

5.7 Extensions of the abductive procedure.

The current SLDNFA procedure can be extended in different ways in order to obtain even more solutions. As a result, the computation trees become larger, the computation is less efficient but additional interesting solutions are obtained. Below an example shows the relevance of the first extension.

Consider the following simplified planning program. An action E initialises a condition p if some initial condition $r(E)$ holds when the action takes place. The same type of action initialises q if a second initial condition $s(E)$ holds. The problem is to find a situation in which both p and q hold. The query is $\leftarrow p, q$. The predicates $action/1$, $r/1$ and $s/1$ are abducible.

$$P = \{p \leftarrow action(E), r(E) \quad q \leftarrow action(E), s(E)\}$$

Intuitively, there are two intersting solutions: $\{action(sk), r(sk), s(sk)\}$ and $\{action(sk_1), r(sk_1), action(sk_2), s(sk_2)\}$.

SLDNFA only generates the second solution while the first is definitely more interesting from the perspective of planning, since it contains less actions. Observe that the substitution $\{sk_1/sk, sk_2/sk\}$ maps the two *action* facts on the same fact in the first solution. Below we extend SLDNFA such that it dynamically tries to *merge* abduced facts. As a result the first solution will also be generated.

The extended SLDNFA has the more interesting completeness property that for any abductive solution Δ, there exists a generated solution Δ' and a skolem substitution σ_{sk} which maps Δ' into Δ, but no facts of Δ' are merged. The cost for this is that the extended procedure crosses a much larger computation tree. Our formulation of the extended algorithm allows a compromise between the improved completeness and the larger computation tree. It provides the opportunity to specify exactly for what abducible predicates the improved completeness should be obtained. The other abducible predicates are dealt with like in SLDNFA. The special abducible predicates will be called *strongly abducible*.

Below we assume the existence of an incomplete logic program P with two different types of undefined predicates: abducible predicates A and strongly abducible predicates SA and $SA \cap A = \phi$. The SLDNFA^o procedure is an extension

of the SLDNFA procedure obtained as follows: the move to $\mathcal{NAG}$ operator and the $\mathcal{NAG}$ operator are executed for both abducible and strongly abducible atoms. The abduction operator is only applied when abducible atoms are selected in positive goals. Two new operators are introduced for the case that a strongly abducible atom A is selected in a positive goal Q. These operators are the following (we assume the existence of $\mathcal{PG}$, $\mathcal{NG}$, $\mathcal{NAG}$ and Δ):

Definition 5.7.1 *Let Q' be derived from Q and some abduced fact from Δ by positive resolution on A and using a positive unifier θ. The* matching operator *produces the substitution θ and the positive goal Q'. Formally:*

$$\mathcal{PG}' = \theta(\mathcal{PG} \setminus \{Q\} \cup \{Q'\})$$
$$\mathcal{NG}' = \theta(\mathcal{NG}), \mathcal{NAG}' = \theta(\mathcal{NAG}), \Delta' = \theta(\Delta)$$

Assume that A is of the form $p(\overline{t})$. Let Q' be derived from Q by abducing A using the skolemising substitution θ. The strong abduction operator *produces the substitution θ, the abduced atom $\theta(A)$, the positive goal Q' and a set of negative goals $\leftarrow \theta(\overline{t}) = \overline{s}$ for each $p(\overline{s}) \in \Delta$. Formally:*

$$\mathcal{PG}' = \mathcal{PG} \setminus \{Q\} \cup \{Q'\}$$
$$\mathcal{NG}' = \mathcal{NG} \cup \{\leftarrow \theta(\overline{t}) = \overline{s} \mid p(\overline{s}) \in \Delta\}$$
$$\Delta' = \Delta \cup \{\theta(A)\}$$
$$\mathcal{NAG}' = \mathcal{NAG}$$

The negative equality goals $\leftarrow \theta(\overline{t}) = \overline{s}$ guarantee that A and the other abduced axioms can never unify.

SLDNFA$^\circ$ differs from SLDNFA in its treatment of strongly abducible atoms, by allowing that either resolution with existing abduced facts is performed, or that a new abduced fact is introduced which is different from the previous ones.

The definitions of refutation, proof tree, computable children, SLDNFA$^\circ$-tree, state and explanation formula remain unaltered. The computable children for a given selection are the same as for pure SLDNFA, except when a positive goal and a strongly abducible atom A is selected. In that case, for each unifiable strongly abduced atom B there is a new child, obtained by positive resolution with B.

Theorem 5.7.1 *The SLDNFA$^\circ$ procedure is sound. It satisfies the same completeness result that was proven for the SLDNFA procedure in theorem 5.6.1. In addition to the assertion (c), the skolem substitution σ_{sk} maps distinct strongly abduced facts of the generated solution Δ' to distinct facts in Δ.*

Proof We must fit in the two new operators into the proof of soundness and completeness of SLDNFA. This involves proving successively the correctness of the two operators, of the proof tree , of the explanation formula and finally of the extended completeness result.

Let Q be a positive goal in which a strongly abducible atom $L_m = p(\overline{t})$ is selected. Given a set of abduced atoms $\{A_1, \ldots, A_g\}$, by positively resolving Q and $A_i = p(\overline{s}_i)$, one computes positive resolvents Q_i and positive unifiers $\theta_i = \sigma_i o \theta'_i$, where θ'_i is the solved form of $L_m = A_i$ and σ_i is the skolemising substitution. For the completeness, the following equivalence is important:

$$\begin{aligned} FEQ(\mathcal{L}) \models \forall(\&(Q) \Leftrightarrow\ & (\theta'_1 \wedge \&(Q'_1) \wedge \theta'_1(A_1)) \vee \ldots \vee \\ & (\theta'_g \wedge \&(Q'_g) \wedge \theta'_1(A_g)) \vee \\ & (\&(Q'_{g+1}) \wedge L_m \wedge \neg\overline{t} = \overline{s}_1 \wedge \ldots \wedge \neg\overline{t} = \overline{s}_g)) \end{aligned}$$

Here $Q'_i = \sigma_i^{-1}(Q_i)$ (σ_i^{-1} is the deskolemising inverse of σ_i). Q_{g+1} is the result of abducing L_m using a skolemising substitution $\theta_{g+1} = \sigma_{g+1}$. Q'_{g+1} is its deskolemisation, i.e. it is the goal obtained by deleting L_m from Q.

The equivalence is easy to obtain. Consider the following tautology :

$$\overline{t} = \overline{s}_1 \vee \ldots \vee \overline{t} = \overline{s}_g \vee (\neg\overline{t} = \overline{s}_1 \wedge \ldots \wedge \neg\overline{t} = \overline{s}_g)$$

By adding this tautology to $\&(Q)$ (as an additional conjunct), then moving its disjunction outside the conjunction and applying equality reduction, we obtain the stated equivalence.

From this equivalence the following implications can easily be derived. They are important for the soundness of the matching operator and strong abduction operator:

$$\begin{aligned} & FEQ(\mathcal{L}) \models \forall(\theta_i(\&(Q)) \Leftarrow \&(Q_i) \wedge \theta_i(A_i)) \qquad (1 \leq i \leq g) \\ & FEQ(\mathcal{L}) \models \forall(\theta_{g+1}(\&(Q)) \Leftarrow \theta_{g+1}(L_m) \wedge \&(Q_{g+1}) \wedge \\ & \qquad\qquad \neg\overline{t}' = \overline{s}_1 \wedge \ldots \wedge \neg\overline{t}' = \overline{s}_g \end{aligned}$$

For the correctness of the proof tree, it suffices to show that the two new operators preserve the correctness, which immediately follows from the implications above. From the correctness of the proof tree, the soundness of the procedure follows directly.

Important for the proof of the completeness is the following observation on the proof tree. Take an SLDNFA$_o$ refutation K which generates Δ' and θ' and let $p(\overline{t}), p(\overline{s})$ be two distinct strongly abduced atoms in Δ'. One easily verifies that if $\overline{t}$ and $\overline{s}$ are positively unifiable, then $\leftarrow \overline{t} = \overline{s}$ belongs to the proof tree. Indeed, assume that $p(\overline{t}^o), p(\overline{s}^o)$ are abduced resp. at step i and j (assume $i < j$) such that $\theta_n o \ldots o \theta_i(p(\overline{t}^o)) = p(\overline{t})$ and $\theta_n o \ldots o \theta_j(p(\overline{s}^o)) = p(\overline{s})$. At time j, the constraint $\leftarrow \theta_{j-1} o \ldots o \theta_i(\overline{t}^o) = \overline{s}^o$ was added to $\mathcal{NG}_j$. In the proof tree, a node is associated to this goal and it is labelled with $\leftarrow \overline{t} = \overline{s}$, where $\overline{t}$ and $\overline{s}$ are ground. This node has as descendants only other negative goals which contain only equality atoms. Due to the correctness

of the proof tree, we find that if $E_1, \ldots, E_k$ are the leaves of the proof tree under the node, then $FEQ(\mathcal{L}) \models \neg\overline{t} = \overline{s} \Leftarrow E_1 \wedge \ldots \wedge E_k$. Due to the deskolemisation Lemma 5.4.1, it follows that $FEQ(\mathcal{L}) \models \forall(\mathcal{D}^{-1}(\neg\overline{t} = \overline{s} \Leftarrow E_1 \wedge \ldots \wedge E_k))$. This turns out to be essential for the proof of the completeness.

The computable children for an SLDNFA$_o$ derivation with a leaf in which a positive goal and a strongly abducible atom are selected, are the extensions $K_1, \ldots, K_g$ obtained by applying the matching operator for each existing unifiable abduced atom and K_{g+1} obtained by applying the strong abduction operator. To prove the correctness of the explanation formula it suffices to show $comp_3(\mathcal{L}, P) \models \forall(state(K) \Leftrightarrow state(K_1) \vee \ldots \vee state(K_g) \vee state(K_{g+1})$. This follows easily from the equivalence formula introduced above, using the same techniques as in Lemma 5.6.1.

Finally, consider the completeness theorem. Assume (θ, Δ) is a consistent abductive solution. We take the same type of model M of $comp_3(\mathcal{L}', P + \Delta)$ as in item (c) of theorem 5.6.1. Proceeding as in the proof of (c), we find a branch K_i of W and a variable assignment V'' such that $M \models V''(Q \wedge \theta \wedge \theta'_{desk} \wedge F_{abd} \wedge F_{ng})$. From this we can derive the existence of a substitution σ' which maps F_{abd} into Δ, and $M \models V''(\sigma')$. The derived σ_{sk} maps Δ' into Δ. We must prove that σ_{sk} maps distinct strongly abducible facts from Δ' to distinct facts of Δ.

Take any pair of distinct strongly abduced facts $p(\overline{t}), p(\overline{s})$ in Δ'. Above we observed that there exist leaves $E_1, \ldots, E_k$ of the proof tree such that $FEQ(\mathcal{L}) \models \forall(\mathcal{D}^{-1}(\neg\overline{t} = \overline{s} \Leftarrow E_1 \wedge \ldots \wedge E_l))$. Let $\overline{t}', \overline{s}'$ be the deskolemisations of $\overline{t}, \overline{s}$ respectively. Observe that F_{ng} contains the deskolemisations of $E_1, \ldots, E_l$. Moreover, since $M \models V''(F_{ng})$, $M \models V''(\mathcal{D}^{-1}(E_i))$. Therefore, $M \models V''(\neg\overline{t}' = \overline{s}')$. Here we use the fact that all variables in $\overline{t}$' and $\overline{s}$' occur in the domain of V''. That is because $var(\{\overline{t}', \overline{s}'\}) \subseteq var(F_{abd})$.

Since $M \models V''(\sigma')$, it holds that $M \models V''(\neg\sigma'(\overline{t}') = \sigma'(\overline{s}'))$ and the terms cannot be identical. □

With respect to the implementation, the addition of all disequality constraints by the strong abduction operator, implies a serious overhead. What happens if they are not added? In that case, a solution may be generated in which the same abduced fact is abduced more than once. However, this necessarily implies that the same solution is generated also in another branch (namely the branch in which this abduced fact is generated only once). So, the completeness result for this procedure remains the same, but more redundant solutions may be created.

For other applications of abduction such as diagnosis, the above obtained completeness result may still be insufficient. An example is found in the faulty lamp problem (section 5.3). Consider the SLDNFA-tree presented in figure 5.2. Observe

that SLDNFA does not generate the solution $\{powerfailure(c1),\ empty(b1)\}$, representing the situation of an empty battery. This does not contradict with the completeness result (c) for SLDNFA. Indeed, the skolem substitution $\{sk = c1\}$ maps the generated solution $\{powerfailure(sk)\}$ into the empty battery solution. Note that under this skolem substitution, the remaining constraint $\leftarrow sk = c1, \neg empty(b1)$ still holds, but for a different reason: $empty(b1)$ holds, thus $\neg empty(b1)$ finitely fails. The application of the skolem substitution does not preserve the fact on which the generated solution depends (namely $sk \neq c1$).

SLDNFA can easily be extended in order to find these solutions. The idea is the following: a constraint containing an atom $sk = t$ is satisfied when $sk \neq t$ is satisfied or when sk equals t and the remainder of the constraint finitely fails. The SLDNFA$_+$ procedure is an extension of the SLDNFA procedure by adding two new operators which apply when an irreducible atom $sk = t$ in a negative goal Q is selected. Note that when SLDNFA's negative resolution operator is applied on such a selection, then nothing happens since the equality reduction of an irreducible equality atom $sk = t$ is $sk = t$ itself.

Definition 5.7.2 *The* assert disequality operator *produces the empty substitution and the negative goal* $\leftarrow sk = t$. *Formally:*

$$\mathcal{NG}' = \mathcal{NG} \setminus \{Q\} \cup \{\leftarrow sk = t\}$$
$$\mathcal{PG}' = \mathcal{PG}, \mathcal{NAG}' = \mathcal{NAG}, \Delta' = \Delta \text{ and } \theta \text{ is } \varepsilon$$

The assert equality operator *does the contrary: it unifies* sk *and* t *and tries to fail the remaining goal. Let* Q' *be the goal obtained by deleting* $sk = t$ *from* Q *and* θ *the positive unifier of* $sk = t$. *The* assert equality operator *produces the substitution* θ *and the negative goal* $\theta(Q')$. *Formally:*

$$\mathcal{NG}' = \theta(\mathcal{NG}) \setminus \{Q\} \cup \{\theta(Q')\}$$
$$\mathcal{PG}' = \theta(\mathcal{PG}), \mathcal{NAG}' = \theta(\mathcal{NAG}), \Delta' = \theta(\Delta)$$

Observe that it makes no sense to apply the new operators on an atomic negative goal $\leftarrow sk = t$. The result of the assert disequality operation is $\leftarrow sk = t$, the result of the assert equality operation is failure.

Definition 5.7.3 *An SLDNFA*$_+$ *refutation* K *for a goal* Q *is a finite SLDNFA*$_+$ *derivation (say of length* n*) which satisfies the same conditions as an SLDNFA refutation. In addition, we require that all constraints in* $\mathcal{NG}_n$ *are atomic irreducible equality goals.*

Example In the faulty lamp example, there is an SLDNFA$_+$ derivation for the goal $\leftarrow faulty\_lamp$ which generates the solution with the empty battery:

$$\mathcal{PG} = \{\leftarrow \underline{faulty\_lamp}\} \qquad \textit{Positive resolution}$$
$$\mathcal{PG} = \{\leftarrow \underline{powerfailure(X)}, \neg backup(X)\} \quad \textit{Abduction}$$

$$\begin{array}{ll}
\mathcal{PG} = \{\leftarrow \underline{\neg backup(sk)}\},\ \Delta = \{powerfailure(sk)\} & \\
 & \textit{Switch to } \mathcal{NG} \\
\mathcal{PG} = \{\Box\},\ \mathcal{NG} = \{\leftarrow \underline{battery(sk,Y)}, \neg empty(Y)\} & \\
 & \textit{Negative Resolution} \\
\mathcal{NG} = \{\leftarrow \underline{sk = c1}, \neg empty(b1)\} & \textit{Assert equality} \\
\mathcal{NG} = \{\leftarrow \neg empty(b1)\} & \textit{Switch to } \mathcal{PG} \\
\mathcal{PG} = \{\leftarrow empty(b1)\},\ \mathcal{NG} = \{\} & \textit{Abduction} \\
\mathcal{PG} = \{\Box\},\ \Delta = \{powerfailure(c1), empty(b1)\} &
\end{array}$$

The definitions of proof tree, computable children, SLDNFA$_+$-tree, state and explanation formula are as for SLDNFA. A derivation K in which an irreducible equality atom is selected, has two computable children, obtained by applying the two new operators. It is easy to see that the SLDNFA$_+$-tree is larger than the corresponding SLDNFA-tree. In fact, examples exist in which SLDNFA$_+$ loops and SLDNFA does not.

Theorem 5.7.2 *SLDNFA$_+$ is sound. It satisfies the same completeness result that was proven for the SLDNFA procedure. In addition to the assertion of item (c) of theorem 5.6.1, if $\leftarrow sk = t$ belongs to $\mathcal{NG}_n$ then $\sigma_{sk}(sk)$ and $\sigma_{sk}(t)$ do not unify.*

Proof As for SLDNFA$_+$, we must fit in the two new operations into the proof of soundness and completeness of SLDNFA. The first step is to derive a correctness result for the assert equality and assert disequality operators. Let Q be a negative goal in which an irreducible equality atom $sk = t$ is selected. Let σ be the skolemising substitution for t and Q' be obtained by deleting $sk = t$ from Q and then applying $\{sk = \sigma(t)\}$. We prove the following equivalence:

$$FEQ(\mathcal{L}) \models Q \Leftrightarrow \forall \overline{Z} : (\neg sk = t) \vee \exists \overline{Z} : (sk = t \wedge \sigma^{-1}(Q'))$$

where $\overline{Z}$ are the variables of t.

The proof goes as follows. The formula below is a tautology:

$$Q \Leftrightarrow (Q \wedge \forall \overline{Z} : \neg sk = t) \vee (Q \wedge \exists \overline{Z} : sk = t)$$

In the first disjunct, Q is subsumed by $\forall \overline{Z} : \neg sk = t$. Because of this and because the existential variables $\overline{Z}$ in the second disjunct do not occur freely in Q, the formula below is still a tautology:

$$Q \Leftrightarrow \forall \overline{Z} : \neg sk = t \vee \exists \overline{Z} : (sk = t \wedge Q)$$

Let Q be of the form $\leftarrow L_1, \ldots, sk = t, \ldots, L_k$ with free variables $\overline{Y} \cup \overline{Z}$. Inside Q, rename $\overline{Z}$ to $\overline{Z}$' and then substitute sk by t. We get as second

disjunct $sk = t \wedge \forall \overline{Y}, \overline{Z}' : L'_1, \ldots, t = t', \ldots, L'_n$, where L'_i, t' are the renamed versions of L_i, t. Observe that since t and t' are renamed copies, applying equality reduction on $t' = t$ yields $\overline{Z}' = \overline{Z}$. It holds that $FEQ(\mathcal{L}) \models \forall(t = t' \Leftrightarrow \overline{Z}' = \overline{Z})$. Replacing $t = t'$ by $\overline{Z}' = \overline{Z}$ and then applying variable elimination on $\overline{Z}'$ yields the desired equivalence.

From this equivalence the following two implications can be derived (using variable elimination and the deskolemisation lemma):

$$FEQ(\mathcal{L}) \models Q \Leftarrow \forall \overline{Z} : \neg sk = t$$
$$FEQ(\mathcal{L}) \models \theta(Q) \Leftarrow Q'$$

The correctness of the proof tree follows easily from these implications. The soundness of SLDNFA$_+$ is a direct consequence. The proof of the correctness of the explanation formula is simple using the equivalence.

Finally, consider the completeness theorem. The construction of σ' and σ_{sk} is the same as in the previous completeness results. We must prove that the disequality constraints $\leftarrow sk = t$ in $\mathcal{NG}_n$ are maintained under σ_{sk}.

Consider any constraint $\leftarrow sk = t$ in $\mathcal{NG}_n$. We must show that $\sigma_{sk}(sk)$ and $\sigma_{sk}(t)$ do not unify. Recall that F_{ng} contains the deskolemisation of the constraint. Let X be $\mathcal{D}^{-1}(sk)$, t' be $\mathcal{D}^{-1}(t)$ and let $\overline{Z}$ be $var(t)$. So F_{ng} contains $\forall \overline{Z} : X = t'$. By the construction of σ_{sk}, it follows that $\sigma_{sk}(sk) \equiv \sigma'(X)$ and $\sigma_{sk}(t) \equiv \sigma'(t')$. Therefore, it suffices to show that $\sigma'(X)$ and $\sigma'(t')$ do not unify.

Since $M \models V''(F_{ng})$, it holds that $M \models V''(\forall \overline{Z} :\leftarrow X = t')$ and since $M \models V''(\sigma')$, it holds that $M \models V''(\forall \overline{Z} :\leftarrow \sigma'(X) = \sigma'(t'))$. Now assume that $\sigma'(X)$ and $\sigma'(t')$ have a unifier ρ. Since X occurs in F_{abd}, $\sigma'(X)$ is ground. Therefore, the domain of ρ is the set of variables $\overline{Z}$. The following identities holds: $\sigma'(X) \equiv \rho(\sigma'(X)) \equiv \rho(\sigma'(t'))$. Because ρ is ground on all variables of $\overline{Z}$, we can extend V'' with assignments $Z_l = \tilde{M}(\rho(Z_l))$ into V'''. Obviously it holds that $M \models V'''(\rho)$. Therefore, $M \models V'''(\rho(\sigma'(t')) = \sigma'(t'))$. Because of this equality and the identities that we just derived, we find that $M \models V'''(\sigma'(X) = \sigma'(t'))$. That is in contradiction with $M \models V''(\forall \overline{Z} : \neg \sigma'(X) = \sigma'(t'))$.

□

Observe that the modifications to SLDNFA in SLDNFA^o and SLDNFA$_+$ stand orthogonal to each other. That is, they can be combined to a new procedure SLDNFA$^o_+$. This procedure is sound, and as a completeness result it can be stated that σ_{sk} preserves the disequality constraints and the disequality of strongly abduced atoms. We obtain a (still primitive) framework of abductive procedures in which a number of parameters can be set in order to fit the abductive procedure to the problem domain under consideration.

5.8 Discussion

We have implemented a prototype of the abductive procedure in Prolog. The prototype was extended to an abductive planner for abductive event calculus by adding a module with a constraint solver for temporal reasoning. This procedure uses SLDNFA^o, with the predicate *happens*/1 as strongly abducible predicate. In chapter 7, the power of the system is shown by applying it to planning and general temporal reasoning problems.

Our experiences with SLDNFA have highlighted the need for an intelligent control strategy. Our implementation uses the straightforward *depth first, left to right* control strategy. In many examples, the system enters an infinite branch of the search tree. A solution for this is to execute the planner according to an iterative deepening regime. Loop detection, intelligent control and intelligent backtracking could be of use to the system.

An interesting feature of SLDNFA (and the planner) is the use of integrity constraints. Algorithm 4.9.1 allows to transform any set of integrity constraints to a normal program. The transformation operates by adding for any integrity constraint IC, the rule $false \leftarrow \neg IC$, transforming these rules to a normal program P' using the Lloyd-Topor transformation [LT84] and adding the literal $\neg false$ to the query. Any sound abductive procedure will generate only solutions in which IC holds. For example, assume that one wants to maintain a database on married couples, represented by a predicate $m/2$. In this database, the constraint holds that the predicate $m/2$ is symmetric:

$$m(X,Y) \leftarrow m(Y,X)$$

This integrity constraint is transformed to :

$$false \leftarrow m(Y,X), \neg m(X,Y)$$

We want to "update" the database with the fact that John (j) just married with Mary (m). So $m/2$ is abducible and the update is formulated by:

$$\leftarrow m(j,m), \neg false$$

Its SLDNFA execution is as follows:

$\mathcal{PG} = \{\leftarrow \underline{m(j,m)}, \neg false\}$	*Abduction*
$\mathcal{PG} = \{\leftarrow \underline{\neg false}\}$, $\Delta = \{m(j,m)\}$	*Switch to* $\mathcal{NG}$
$\mathcal{PG} = \{\Box\}$, $\mathcal{NG} = \{\leftarrow \underline{false}\}$	*Negative resolution*
$\mathcal{NG} = \{\leftarrow \underline{m(Y,X)}, \neg m(X,Y)\}$	*Switch to* $\mathcal{NAG}$ *and* $\mathcal{NAG}$ *operation*
$\mathcal{NG} = \{\leftarrow \underline{\neg m(m,j)}\}$, $\mathcal{NAG} = \{(" \leftarrow m(Y,X), \neg m(X,Y)", "m(Y,X)", \{m(j,m)\})\}$	*Switch to* $\mathcal{PG}$

$\mathcal{PG} = \{\Box, \leftarrow \underline{m(m,j)}\}$, $\mathcal{NG} = \{\}$ *Abduction*
$\mathcal{PG} = \{\Box\}$, $\Delta = \{m(j,m), \underline{m(m,j)}\}$ *$\mathcal{NAG}$ operation*
$\mathcal{NG} = \{\leftarrow \underline{\neg m(j,m)}\}$, $\mathcal{NAG} = \{(" \leftarrow m(Y,X), \neg m(X,Y)", "m(Y,X)",$
$\{m(j,m), m(m,j)\})\}$ *Switch to $\mathcal{PG}$*
$\mathcal{PG} = \{\Box, \leftarrow \underline{m(j,m)}\}$, $\mathcal{NG} = \{\}$ *Abduction*
$\mathcal{PG} = \{\Box\}$

The generated "update" is $\{m(j,m), m(m,j)\}$. A few observations about this refutation are in order. First, it shows that even for simple integrity constraints, it is essential that the abductive procedure can deal with non-ground abducible atoms in negative goals. Second, SLDNFA operates as an integrity recovery method: not only the integrity constraints are checked but additional assertions are made in order to restore the integrity. Third, looking at the execution, it turns out that SLDNFA verifies the original integrity constraint in a bottom up way: an instance of the body of the rule is found and this fires the abduction of the head of the rule. A subject for future research is to compare with special purpose procedures (such as the one in [SK88]).

In [CTT91], an abductive procedure is presented which, for a given hierarchical normal abductive program P and query $\leftarrow Q$, derives an *explanation formula* E equivalent with Q under the (2-valued) completion of P:

$$comp(P) \models (Q \Leftrightarrow E)$$

This is done by repeatedly substituting atoms of defined predicates by the equivalent part in their if-and-only-if definition, until no defined atoms are left over. Thus, the explanation formula is built of abducible predicates and equality only. It characterises all abductive solutions in the sense that for any set Δ of abducible atoms, Δ is an abductive solution iff it satisfies E.

SLDNFA can be considered as a procedure for rewriting goals using the definition of the defined predicates. An advantage of the procedure in [CTT91] is that it also applies for non-ground negative literals. On the other hand, observe that in the case of a recursive predicate, repeated naive rewriting of a defined atom by its definition necessarily goes into a loop. SLDNFA can avoid this (in many cases) by checking the consistency of generated equality and disequality atoms (this is done implicitly in the resolution steps) and eliminating an inconsistent branch. Even when provisions for equality would be built into the procedure in [CTT91], there would still be the problem that if the computation tree contains an infinite branch, then the explanation formula cannot be computed. SLDNFA on the other hand investigates the tree branch per branch and, using an iterative deepening regime, will ultimately find all solutions in finite branches.

Another related procedure has been presented in [GL90b]. This belief revision procedure tries to construct an SLDNF-refutation for a given goal by adding facts to the program (as in SLDNFA) to succeed positive goals and by deleting clauses

of the program to fail negative goals. We believe that there is a big conceptual gap between this procedure and SLDNFA. They are in general not applicable in the same context. Whether in a given context the procedure in [GL90b] or SLDNFA is applicable depends totally on the reliability of the general domain knowledge which is formulated in the clauses of the program. When reliable, as in planning, no clauses should be retracted.

A remaining restriction of SLDNFA is its inability to deal with non-ground negative atoms (the problem of floundering negation). Here is an intriguing relationship with the view of *negation by failure as abduction* ([Esh88], [KM90a]). In this view, the problem with non-ground negative atoms is a subproblem of the problem with non-ground abducible literals. In [KM90a], the authors indicate that the methods which they are developing for non-ground abducible literals will also solve the problem with floundering negation. Strong indications exist that the techniques incorporated in SLDNFA can solve the problem of floundering negation for positive goals but not for negative goals[7].

SLDNFA is not only sound wrt the 3-valued completion semantics (and direct justification semantics) but also wrt the more fine-grained semantics of partial justification and justification semantics under *FEQ*. This follows directly from theorem 4.3.3. As mentioned in section 5.5, SLDNFA is not sound wrt 2-valued completion semantics. Indeed, SLDNFA may generate solutions which are not consistent with $comp(P)$. A trivial example is the incomplete program with normal clause $p \text{:-} \neg p$ and undefined predicate r. The goal $\leftarrow r$ is solved by $\Delta = \{r\}$, but $comp(P + \Delta)$ is inconsistent. The same example also shows that SLDNFA is in general not sound wrt the generalised stable semantics [KM90b]: the set $\{r\}$ cannot be completed to a generalised stable model of P. The example shows that to build a sound abductive procedure for 2-valued completion semantics or generalised stable semantics, consistency checking is necessary. A sound abductive procedure has been developed for generalised stable semantics [SI92]. The procedure is an extension of the abductive procedure of [KM90a] with special consistency checking techniques for dealing with predicates like p which are possibly looping over negation. In the above example, the procedure would fail because p cannot be given a consistent truth value. Since SLDNFA does not bother about predicates like p, the procedure of [SI92] must traverse a larger search space than SLDNFA. In general, this extra computation may be costly. Although we do not believe that the efficiency of a procedure should be taken as an argument in favour of a semantics, it is nevertheless extremely convenient that wrt to 3-valued completion semantics or (direct) (partial) justification semantics, this extra consistency checking is unnecessary. Note also that for overall consistent logic programs, SLDNFA is sound wrt generalised stable model semantics.

[7] These "strong indications" are that the proof of the correctness of the switch to $\mathcal{NG}$ operator in proposition 5.4.6 does not depend on the fact that the negative literal is ground. The proof of the correctness of the switch to $\mathcal{PG}$ operator (same proposition) definitely depends on the groundness of the negative literal.

By setting a number of parameters (i.e. strongly abducible predicates, special treatment of disequality constraints) SLDNFA$^{o}_{+}$ can be tuned to the application under consideration. One common element of the procedures in the framework is that they try to minimize Δ. This is not always desirable. One example is the intentional update problem for databases. Given is a database D containing a set of facts of *base* predicates, a logic program P defining a set of *view* predicates and a formula F which represents an intentional update: we want to modify the database D in such a way that $D + P \models F$. The abductive view on this problem presented in [KM90a] is that the base predicates in D are abducible and that we search an abductive solution D' such that $P + D' \models F$. SLDNFA is not directly suited for this. The problem is that the desired solutions are not the minimal databases satisfying F, but the databases D' which are as close as possible to D: the set $D' \setminus D \cup D \setminus D'$ should be minimal. As a completeness result for an abductive database update problem, we expect that for any database D'' which is a solution for the intentional database update, there exists a generated database D' which is closer to D; i.e. $D' \setminus D \subseteq D'' \setminus D$ and $D \setminus D' \subseteq D \setminus D''$. Or, every positive or negative fact in D which does not occur in D', neither occurs in D''.

Despite this, SLDNFA can be used for database updating but using a meta-approach, an idea already used in [Bry90][8]. The idea is to describe the new database in function of the old one and a set of $insert/1$ and $retract/1$ facts. We sketch the approach. Below, a database consists of a set Db of ground facts of *base predicates* and an incomplete program P defining a set of *view* predicates in terms of the base predicates. We assume -without loss of generality- that the database does not contain integrity constraints[9]. An intentional database update is a FOL formula F which is to be implied by $P + \Delta(Db)$ under some semantics, where $\Delta(Db)$ is the database obtained by retracting some of the existing facts of Db and inserting some new ones. We build a meta-program P_m in the following way:

- The intentional update $\leftarrow F$ is transformed using algorithm 4.9.1 to a normal query $\leftarrow F'$ and a logic program P_F.
- For each atom $A \in Db$, P_m contains $db(A)$:- .
- For each normal clause A :- $B \in P + P_F$, P_m contains the fact $clause(A, B)$:- . Below we assume that the last argument of the body of any normal clause is always the atom $true$.
- The following vanilla meta-program describes the predicates $new/1$ and $old/1$:

[8] [Bry90] proposes a meta-theory which, executed by a model generator, generates an intentional update.

[9] If there are integrity constraints, then they should first be compiled into a definition for $false$ and the literal $\neg false$ should be added to the intentional update.

$old(F) \text{ :- } base(F), db(F)$
$old(F) \text{ :- } clause(F, B), old(B)$
$old(true) \text{ :-}$
$old((F, B)) \text{ :- } old(F), old(B)$
$old(\neg F) \text{ :- } \neg old(F)$

$new(F) \text{ :- } base(F), db(F), \neg retract(F)$
$new(F) \text{ :- } base(F), insert(F)$
$new(true) \text{ :-}$
$new(F) \text{ :- } clause(F, B), new(B)$
$new((F, B)) \text{ :- } new(F), new(B)$
$new(\neg F) \text{ :- } \neg new(F)$

We obtain an incomplete program with definitions for $base/1, clause/2, db/1, old/1$ and $new/1$, and undefined predicates $insert/1$ and $retract/1$. Note that the definitions for $old/1$ and $new/1$ are almost literal copies of the standard vanilla meta interpreter. When SLDNFA (or one of its variants) solves the goal $\leftarrow new(F')$, it returns a set Δ of $insert/1$ and $retract/1$ facts such that $P_m + \Delta \models F$ (under 3-valued completion semantics, (direct) (partial) justification semantics). By using SLDNFA^o under an iterative deepening approach allowing more and more insert and retract facts to be abduced, one will ultimately find a minimal solution (under the assumption that SLDNFA does not flounder). It was observed in [Bry90] that the meta-approach has several interesting features such as the statement of dynamic integrity constraints as in

$$\forall X, X_1, Y : old(salary(X, Y)) \wedge new(salary(X_1, Y) \Rightarrow X_1 \geq X$$

and intentional updates referring to both old an new state, such as:

$$\forall X : old(professor(X)) \wedge old(teach(X, lp)) \Rightarrow new(qualified\_for\_tenure(X))$$

A deeper exploration of this intriguing issue is beyond the scope of this work. Whether SLDNFA (or a partial evaluation of it) is sufficiently efficient remains an open question to us and is subject to future research.

5.9 Summary

SLDNFA is an abductive procedure extending SLDNF for normal incomplete programs. Its most distinguished property is that it does not flounder on non-ground abducible atoms, as is the case with procedures such as [KM90a] and [SI92]. This property is essential for application in many, perhaps most problem domains: the examples in this chapter illustrated that even for very simple temporal reasoning problems in incomplete event calculus, for simple diagnosis problems and for simple

database problems, the floundering problem pops up. Other procedures developed specifically for dealing with the floundering problem are either not formalised and proven correct as in [Esh88], [Sha89], or only correct for a subclass of the normal logic program formalism as in [Mis91b, Mis91a]. Moreover, compared with these procedures, SLDNFA incorporates an improved behaviour for negative abductive goals.

The soundness of SLDNFA wrt 3-valued completion semantics and (direct) (partial) justification semantics has been proven: for a generated solution Δ, it holds that both $P + \Delta$ is consistent and entails the query. This implies the soundness of SLDNFA wrt the constructive definition view (chapter 4).

Another contribution of this work is the formulation of new general completeness criteria for abductive procedures. It was proven that the completeness criterion to be satisfied by an abductive procedure is in general problem dependent: examples in sections 5.7 and 5.8 illustrate this for planning, diagnosis and database updating. To cope with this problem, extensions of SLDNFA were developed. This results in a (still simple) family of abductive procedures, SLDNFA$^{o}_{+}$, in which a number of parameters can be set to tune the procedure to the application under consideration.

We have proven that for the given completeness criteria, specific instances of the SLDNFA$^{o}_{+}$ family satisfy them wrt 3-valued completion semantics and (direct) (partial) justification semantics, under the condition that a finite SLDNFA-tree is generated. This is a strong condition, hiding for example the floundering on non-ground negative atoms and looping, but we expect that, as for SLDNF, less restrictive conditions can be found (e.g. allowedness and strictness) under which SLDNFA satisfies the completeness criteria.

In chapter 1, we argued that one advantage of a declarative logic is that different procedures can be defined on it and can operate on the same logic specification. So far the incomplete logic program formalism has been associated uniquely with abduction as procedural paradigm (this explains the terminology *abductive logic program*). In section 5.6.1, we took a start to decouple the incomplete logic program formalism and abduction, by showing that SLDNFA can be used for sound *deduction* in the formalism. This, together with the declarative view on (incomplete) logic programs such as the constructive definition view is a step towards turning the formalism into a full-fledged declarative logic.

Chapter 6

A translation of $\mathcal{A}$ to incomplete situation calculus.

6.1 Introduction

Recently, [GL92] introduced a new temporal language $\mathcal{A}$ which allows to represent a number of well-known benchmark problems involving incomplete temporal knowledge. They proposed a sound but incomplete transformation to extended logic programs, programs with both negation as failure and classical or explicit negation [GL90a]. We present a transformation from $\mathcal{A}$ domain descriptions to incomplete programs with FOL axioms (section 6.3). The proposed transformation maps an $\mathcal{A}$ domain description to an incomplete situation calculus and is proven to be sound and complete. The chapter can be seen as a -successful- case study in the representation off incomplete knowledge.

A secondary goal is to illustrate how an abductive procedure can be useful for automated reasoning with incomplete programs and integrity constraints. That an abductive procedure can be used for *explanation* of some observation is well-known from [Pei55], [Sha89]. It is less known that an abductive procedure can also be used for deduction and for proving consistency of a theory. In section 4.6, we argued that an abductive procedure can be used to prove consistency of a theory. In section 5.6 we argued that an abductive procedure can be used for deduction in an incomplete logic program. In section 6.4, we illustrate this by applying SLDNFA to solve distinct computational tasks involving complete and incomplete knowledge.

Independent from the work presented here, another approach has been developed for translating $\mathcal{A}$ domain descriptions to a logic program formalism. This

work, described in [Dun93], maps $\mathcal{A}$ to a special purpose logic, based on the logic program formalism but with a new semantics, adapted to this specific application. The semantics is a variant of the completion semantics. It is shown that in this logic a partial deduction procedure can be used to generate abductive solutions consisting of goals on the initial situation. Our work differs at four important points from [Dun93]: we map $\mathcal{A}$ to a general purpose logic, i.e. standard abductive logic programming, thus showing that a special purpose treatment is unnecessary; our transformation is simpler than in [Dun93] and [GL92]; we use a general purpose abductive procedure instead of a partial deduction procedure; we show how this procedure can be useful to implement other computational paradigms such as deduction and satisfiability proving. There are other differences: while [Dun93] applies (a predicate extension of) $\mathcal{A}$ in the context of database updating, we spend more attention to the issue of temporal reasoning itself, for example by analysing the relationship between backward persistence and forward persistence axioms in our approach and in [GL92], and by showing how $\mathcal{A}$ could be extended for indeterminate actions.

The chapter is structured as follows. In section 6.2, we recall the language $\mathcal{A}$ and its semantics. In section 6.3, the transformation from $\mathcal{A}$ to situation calculus programs is presented and the soundness and completeness is proved. In section 6.4, the use of abduction for explanation and deduction is illustrated. Section 6.5 gives a comparison between our transformation and the transformation in [GL92]. Section 6.6 compares our work with [Dun93] and 6.7 discusses other related work. A short paper on this subject will be published as [DD93b].

6.2 The temporal language $\mathcal{A}$

The language $\mathcal{A}$ [GL92] allows to describe relationships between fluents (= time dependent properties of the world) and actions. $\mathcal{A}$ is a propositional language: both fluents and actions are represented by propositional symbols. Two types of expressions occur. A v-proposition describes the value of a fluent after a (possibly empty) sequence of actions. Its syntax is as follows:

f **after** $a_1; \ldots; a_n$

Here $a_1; \ldots; a_n$ is a sequence of action symbols and f is a fluent expression: a positive or negative literal containing a fluent. The expression means that f is true after executing the sequence of actions $a_1; \ldots; a_n$. When the sequence of actions is empty (n=0), the v-proposition describes the initial situation. Instead of f **after** , one usually writes:

Initially f

An e-proposition describes the effect of actions on the fluents. It has the form:

a **causes** f **if** $p_1, \ldots, p_n$

where $f, p_1, \ldots, p_n$ are fluent expressions and a is an action symbol. The expression means that if $p_1, \ldots, p_n$ are true, then the effect of a on the current situation is that f becomes true. $p_1, \ldots, p_n$ are called preconditions. When n=0, one writes:

a **causes** f.

A domain description is a set of v- and e-propositions.

Example We recall the Yale Turkey Shooting problem (YTS) as formulated in [GL92]. The fluents are *loaded*, *alive*; the action names are *shoot*, *wait* and *load*. The domain description D_0 contains the following propositions:

Initially *alive*
Initially $\neg$*loaded*
load **causes** *loaded*
shoot **causes** $\neg$*alive* **if** *loaded*
shoot **causes** $\neg$*loaded*

Example The Murder Mystery domain D_1 is a variant of YTS, obtained by substituting

$$\neg alive \textbf{ after } shoot; wait$$

for **Initially** $\neg$*loaded* in D_0. This is a prototypical postdiction problem: we want to obtain the conclusion that initially the gun is loaded.

The semantics for $\mathcal{A}$ is defined as follows. A *state* is a set of fluent names and describes a possible state of the world. Given a fluent symbol f and a state σ, f holds in σ if $f \in \sigma$, otherwise $\neg f$ holds in σ. A transition function Φ maps pairs (a, σ) of action symbols a and states σ into the set of states. Φ describes how a situation changes under application of an action a. A structure M is a pair (σ_0, Φ), where σ_0 represents the initial state and Φ the transition function. $M^{a_1;\ldots;a_m}$ denotes the state $\Phi(a_m, \Phi(a_{m-1}, \ldots, \Phi(a_1, \sigma_0)\ldots))$. A v-proposition

$$f \textbf{ after } a_1; \ldots; a_m$$

holds in a structure (σ_0, Φ) iff f holds in $M^{a_1;\ldots;a_m}$.

Definition 6.2.1 *A structure $M = (\sigma_0, \Phi)$ is a model of a domain expression D if and only if the following rules are satisfied:*

- *Each v-proposition f* **after** *$a_1; \ldots; a_m$ holds in M.*

- *For any state σ, fluent symbol f and action a, if there exists an e-proposition a* **causes** f **if** $p_1, \ldots, p_n$ *such that $p_1, \ldots, p_n$ hold in σ, then f holds in $\Phi(a, \sigma)$. If there is an e-proposition a* **causes** $\neg f$ **if** $p_1, \ldots, p_n$ *such that $p_1, \ldots, p_n$ hold in σ, then $\neg f$ holds in $\Phi(a, \sigma)$. Otherwise, f holds in $\Phi(a, \sigma)$ iff f holds in σ.*

A domain description is called *consistent* if it has a model. We introduce a new notion, *e-consistency*. A domain description is *e-consistent* if the set of e-propositions of D is consistent. There is a simple necessary and sufficient condition for a domain description to be e-consistent.

Lemma 6.2.1 *A domain description D is e-consistent iff for each pair of rules*

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \textit{ and } a \textbf{ causes } \neg f \textbf{ if } p_{n+1}, \ldots, p_m$$

in D, there exists an i and j such that p_i is the complement of p_j.

This condition is satisfied when the complementary literals are found in the bodies of the two rules but also when they appear in the body of one rule, as in *shoot* **causes** *alive* **if** *loaded*, $\neg$*loaded*. Such a rule has an inconsistent body. It can never be applied and can never cause an inconsistency.

In [Dun93], an action a for which there exists a pair of rules

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \text{ and } a \textbf{ causes } \neg f \textbf{ if } p'_1, \ldots, p'_m$$

such that no complement of a literal in the first is contained in the second is called self-contradicted. [Dun93] contains a similar proposition as lemma 6.2.1.

Proof Assume that for some pair of rules:

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \text{ and } a \textbf{ causes } \neg f \textbf{ if } p_{n+1}, \ldots, p_m$$

no complementary literals occur in $\{p_1, \ldots, p_m\}$. This set contains positive and negative literals. Define the state σ consisting of only the positive literals in the body of the two rules. Then obviously each p_i holds in σ. $\Phi(a, \sigma)$ is not consistently defined because both f and $\neg f$ should hold in $\Phi(a, \sigma)$.

Vice versa, one easily verifies that when the syntactical condition in the lemma is satisfied, the definition of a model of a domain description gives a consistent description of a transition function Φ. □

Definition 6.2.2 (Non-inertial) *We say that a fluent f is non-inertial under an action a in a state σ iff there exists an e-proposition*

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \textit{ or } a \textbf{ causes } \neg f \textbf{ if } p_1, \ldots, p_n$$

such that $p_1, \ldots, p_n$ hold in σ. Otherwise, it is inertial under action a.

[GL92] observes that the e-propositions of a domain D completely determine the transition function Φ. Below proposition 6.2.1 gives a precise and deterministic characterisation of Φ, under the condition that D is e-consistent.

Proposition 6.2.1 *Let D be e-consistent. For any state σ, action a and positive fluent symbol f: $f \in \Phi(a, \sigma)$ iff*

- *f holds in σ and f is inertial under a in state σ, or*
- *there exists an e-proposition a* **causes** *f* **if** *$p_1, \ldots, p_n$ such that $p_1, \ldots, p_n$ hold in σ.*

The proof is straightforward.

Note that if the condition of e-consistency is not satisfied, then the description of Φ in the proposition does not correspond to the description of Φ in definition 6.2.1. Indeed, for f, a and σ such that for e-propositions

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \text{ and } a \textbf{ causes } \neg f \textbf{ if } p'_1, \ldots, p'_m$$

the fluents $p_1, \ldots, p_n, p'_1, \ldots, p'_m$ hold in σ, the second item of definition 6.2.1 requires that both $f \in \Phi(a, \sigma)$ and $f \notin \Phi(a, \sigma)$. This is a contradiction. On the other hand, proposition 6.2.1 requires that $f \in \Phi(a, \sigma)$ since the first item applies. Because the transformation proposed in next section implements the formulation of proposition 6.2.1 rather than that of definition 6.2.1, it will be complete only for e-consistent domains.

A v-proposition Q is *entailed* by a domain description D iff Q holds in each model of D. A domain description is called *complete* if it has a unique model. The YTS domain and Murder Mystery domain are examples of complete domain descriptions. Since they share their e-propositions, their models have an identical transition function Φ which maps tuples $(wait, \sigma)$ on σ, $(load, \sigma)$ on $\sigma \cup \{loaded\}$, and maps $(shoot, \sigma)$ on $\sigma \backslash \{alive, loaded\}$ if $loaded \in \sigma$, otherwise on σ. The model M_0 of D_0 has initial situation $\{alive\}$. The model M_1 of D_1 has initial situation $\{alive, loaded\}$. An *incomplete* domain description is obtained by dropping the v-proposition **Initially** *alive* from the Murder Mystery domain. One additional model is the structure with transition function Φ but with initial situation $\{\}$.

$\mathcal{A}$ provides only restricted expressivity: the language is only propositional, no relationships between fluents can be defined, no indeterminate events are allowed. Nevertheless, $\mathcal{A}$ allows to formalise several interesting domains. This and its clear semantics makes the language interesting for experiments as in [GL92], [Dun93] and in this chapter.

6.3 Translation to Incomplete Logic Programs

In this section we present a general translation from an $\mathcal{A}$ domain description D to an incomplete logic program with integrity constraints. The transformation pro-

duces programs in situation calculus style. Traditionally, two options are available to represent a fluent f in a logic formalism: by a predicate $f(s)$ or by $Holds(f, s)$ where s is a state argument. Then $\neg f$ is translated to $\neg f(s)$ or $\neg Holds(f, s)$. The two approaches are equivalent but the meta-approach has the advantage that the frame axiom can be stated for all fluents at once, whereas in the first approach one frame axiom per fluent predicate is needed. As in [GL92], we use $Holds/2$. The first order language $\mathcal{L}_D$ contains the predicate symbols $Holds/2$, $Noninertial/3$ and $Initially/1$. Each fluent and action symbol occurs in $\mathcal{L}_D$ as a constant. In addition, there is a constant s_0 to denote the initial state and a functor $Result/2$: $Result(a, s)$ denotes the state obtained by applying action a on state s. In the sequel, we will use $Result[a_1; \ldots; a_n, s]$ as a shorthand notation for $Result(a_n, Result(a_{n-1}, \ldots Result(a_1, s) \ldots))$. For $n = 0$, this denotes s.

$\mathcal{A}$ allows uncertainty on the initial state. Correspondingly, the incomplete program comprises one undefined predicate, $Initially/1$. The translation maps a domain description D to a theory πD consisting of an incomplete logic program P_D and a set of FOL axioms IC_D. P_D is defined as follows:

- Initialisation:
$$Holds(F, s_0) \text{ :- } Initially(F) \tag{6.1}$$

- Law of Inertia:
$$Holds(F, Result(A, S)) \text{ :- } Holds(F, S), \neg Noninertial(F, A, S) \tag{6.2}$$

- For each e-proposition a **causes** f **if** $p_1, \ldots, p_m, \neg p'_1, \ldots, \neg p'_n$ with f, p_i and p'_j positive literals:
$$\begin{aligned} Holds(f, Result(a, S)) \text{ :- } & Holds(p_1, S), \ldots, Holds(p_m, S), \\ & \neg Holds(p'_1, S), \ldots, \neg Holds(p'_n, S) \end{aligned} \tag{6.3}$$
As in [GL92], we introduce the convention that when f is a negative literal $\neg f'$, $Holds(f, t)$ is used as a textual denotation for $\neg Holds(f', t)$. This handsome convention allows us to say that a **causes** f **if** $p_1, \ldots, p_n$ is translated to the clause:
$$Holds(f, Result(a, S)) \text{ :- } Holds(p_1, S), \ldots, Holds(p_n, S)$$
without considering the sign of the literals p_i. Be aware that a program should never contain literals of the form $Holds(\neg f, S)$, and that when these literals are found in this chapter, they always stand for $\neg Holds(f, S)$.

- For each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ with f a positive or negative fluent literal:
$$Noninertial(|f|, a, S) \text{ :- } Holds(p_1, S), \ldots, Holds(p_n, S) \tag{6.4}$$
For a fluent symbol f, $|f|$ and $|\neg f|$ both denote the term f.

The set of FOL axioms IC_D is defined as follows:

- For each v-proposition f **after** $a_1; \ldots; a_n$ (n $\geq$ 0):

$$Holds(f, Result[a_1; \ldots; a_n, s_0]) \tag{6.5}$$

  with the same syntactic convention on $Holds/2$ as above.

Example The domain description D_0 for the YTS problem is transformed to:

$$Holds(F, s_0) \text{ :- } Initially(F)$$
$$Holds(F, Result(A, S)) \text{ :- } Holds(F, S), \neg Noninertial(F, A, S)$$
$$Holds(loaded, Result(load, S)) \text{ :-}$$
$$Noninertial(loaded, load, S) \text{ :-}$$
$$Noninertial(loaded, shoot, S) \text{ :-}$$
$$Noninertial(alive, shoot, S) \text{ :- } Holds(loaded, S)$$

$$Holds(alive, s_0)$$
$$\neg Holds(loaded, s_0)$$

The clause $Noninertial(loaded, load, S)$:- may be dropped from this program without effect on the semantics of Holds/2. In general, all $Noninertial/3$ rules for initiating effects of actions may be dropped, without effect on the semantics of Holds/2.

πD_0 strongly resembles the YTS solution in [AB90]. They propose a Prolog program analogous to the one obtained from πD_0 by substituting the program clause:

$$Holds(alive, s_0) \text{ :-}$$

for the program clauses:

$$Holds(F, s_0) \text{ :- } Initially(F)$$
$$Noninertial(loaded, load, S) \text{ :-}$$

and the two FOL axioms of πD_0. Note that the resulting program entails the two FOL axioms. [AB90] proves that the resulting program is acyclic. The same holds for πD_0, and in fact for all transformed domain descriptions:

Proposition 6.3.1 *The translation πD of any domain description D is acyclic.*

Proof A slight modification of the level mapping proposed in [AB90] for the YTS program applies for all domain descriptions. For all ground terms t, let $|t|_{Result}$ denote the number of occurrences of the functor $Result/2$ in t. We define $|.|$ for all ground terms t, a and s as follows:

$$|Initially(t)| = 0$$
$$|Holds(t,s)| = 2 \times |s|_{Result} + 1$$
$$|Noninertial(t,a,s)| = 2 \times |s|_{Result} + 2$$

One easily verifies that $|.|$ is a level mapping. □

A transformation such as π can be considered correct if the set of entailed formulas are equivalent. In [GL92] a translation π is defined to be sound iff for each domain D and v-proposition Q, if $\pi D \models \pi Q$ then D entails Q. π is defined to be complete if the reverse holds: if D entails Q then $\pi D \models \pi Q$.

We will prove the soundness and completeness of π wrt 3-valued completion semantics. However, π is sound and complete wrt any of the following semantics: 2-valued and 3-valued (direct) (partial) justification semantics with FEQ, 2-valued and 3-valued (partial) justification semantics with FEQ and $SDCA$, 2-valued and 3-valued completion semantics [CTT91], generalised stable semantics [KM90b] and the generalised well-founded semantics [PAA91b]. This follows from the fact that P_D is acyclic, each clause has the property that all variables of the body occur in the head, and each formula of IC_D contains no variables. For such theories, proposition C.2 in appendix C proves that the sets of entailed ground literals wrt to any of the above semantics are identical.

The translation πD of a domain description contains two defined predicates, $Holds/2$ and $Noninertial/3$. The completed definition of $Holds/2$ is of the form:

$$\forall F,T : Holds(F,T) \leftrightarrow E_1 \vee E_2 \vee \ldots \vee E_m \tag{6.6}$$

with:

$$\begin{array}{ll} E_1 = & T = s_0 \wedge Initially(F) \\ E_2 = & \exists A,S : T = Result(A,S) \wedge \\ & \quad Holds(F,S) \wedge \neg Noninertial(F,A,S) \\ E_i (i>2) = & \exists S : F = f \wedge T = Result(a,S) \wedge \\ & \quad Holds(p_1,S) \wedge \ldots \wedge Holds(p_n,S) \end{array}$$

such that precisely for each e-proposition a **causes** f **if** $p_1,\ldots,p_n$ with f a positive literal, there exists one disjunct E_i $(i>2)$ in the completed definition. If we map S on state σ, the $Result$ functor on Φ and $Holds(F,S)$ to $F \in \sigma$ then the completed definition of $Holds/2$ is similar to proposition 6.2.1.

The completed definition of $Noninertial/3$ is of the form:

$$\forall A,F,S : Noninertial(F,A,S) \leftrightarrow E_1 \vee \ldots \vee E_m \tag{6.7}$$

with:

$$E_i = \quad A = a \wedge F = |f| \wedge Holds(p_1,S) \wedge \ldots \wedge Holds(p_n,S)$$

such that precisely for each e-proposition a **causes** f **if** $p_1,\ldots,p_n$ (f positive or negative), there exists one corresponding disjunct in the completed definition. This formula is the counterpart of definition 6.2.2: the formulation is almost identical apart from the fact that $Holds(p_i,S)$ should be replaced by "p_i holds in S".

Definition 6.3.1 *Let M be any interpretation of $\mathcal{L}_D$, x a domain element of M.*

$$state_M(x) = \{f \mid f \text{ is a positive fluent symbol and } M \models Holds(f, x)\}$$

Note that $state_M(x)$ denotes a state as in section 6.2, and a transition function Φ can be applied on it. Note also that $f \in state_M(x)$ is equivalent with $M \models Holds(f, x)$.

Theorem 6.3.1 (soundness) *Let D be a domain description. For any v-proposition Q, if $\pi D \models \pi Q$ then D entails Q.*

Proof Nothing is to be proved when D is inconsistent. So, assume D is consistent. It suffices to show that for each model $M_a = (\Phi, \sigma_0)$ of D and for each v-proposition Q, there exists a model M of πD such that Q holds in (Φ, σ_0) iff $M \models \pi Q$. If then πQ holds in all models of πD, then also in these models corresponding to models of D. Hence, Q holds in all models of D.

We construct a Herbrand model M. $HU(\mathcal{L}_D)$ denotes the Herbrand universe. The basic idea is simple: we define M such that for each state term $s = Result[a_1; \ldots; a_n, s_0]$, $state_M(s) = M^{a_1;\ldots;a_n}$ and such that the atom $Noninertial(a, f, s)$ is true whenever f is non-inertial under a in the state $state_M(s)$. Things are slightly complicated due to the fact the πD is not a sorted program and ill-sorted atoms may occur in the model. For that reason, we extend D to D' by allowing each term $t \in HU(\mathcal{L}_D)$ as a fluent symbol and as an action symbol. Note that non-original fluents t_f (symbols of $HU(\mathcal{L}_D)$ which are not fluent symbols in D) and non-original actions t_a do not occur in the e-propositions. This implies that a non-original fluent t_f of D' always remains as in the initial state; a non-original action t_a has no effect on a state (like the action *wait* in the YTS D_0). (Φ, σ_0) can easily be extended to a model (Φ', σ_0') of D'. Define $\sigma_0' = \sigma_0$. Let σ' be any state of D', consisting of original fluents σ and new fluents σ_n. Extend Φ to Φ' in the following way:

$$\Phi'(t_a, \sigma') = \sigma' \qquad \text{for } t_a \text{ a non-original action}$$
$$\Phi'(t_a, \sigma') = \Phi(t_a, \sigma) \cup \sigma_n \qquad \text{for } t_a \text{ an original action}$$

One easily verifies that (Φ', σ_0') is a model of D'. Moreover, for any v-proposition Q based on the original language of D, Q holds in (Φ', σ_0') iff Q holds in (Φ, σ_0).

Next we associate to (Φ', σ_0') a Herbrand model M of $\pi D = \pi D'$. With any term $t_s \in HU(\mathcal{L}_D)$ we associate a specific state of D'. Below we call a term t_s an empty-state term if $t_s \neq s_0$ and $t_s \neq Result(t_1, t_2)$ for some t_1, t_2. For any term $t_s \in HU(\mathcal{L}_D)$, $state_M(t_s)$ is constructed as follows:

$$
\begin{array}{ll}
t_s = s_0 & \Rightarrow state_M(t_s) = \sigma_0 \\
t_s \text{ is an empty-state term} & \Rightarrow state_M(t_s) = \phi \\
t_s = Result[t_1; \ldots; t_n, s_0] & \Rightarrow state_M(t_s) = \Phi'[t_1; \ldots; t_n, \sigma_0] \\
t_s = Result[t_1; \ldots; t_n, t_0], \; t_0 \text{ an empty-state term} & \\
 & \Rightarrow state_M(t_s) = \Phi'[t_1; \ldots; t_n, \phi]
\end{array}
$$

M is defined as follows:

$$
\begin{array}{l}
\{Holds(t_f, t_s) \mid t_f \in state_M(t_s)\} \cup \\
\{Noninertial(t_f, t_a, t_s) \mid t_f \text{ is non-inertial under } t_a \\
\qquad\qquad\qquad\qquad \text{in } state_M(t_s)\} \cup \\
\{Initially(t_f) \mid t_f \in \sigma_0\}
\end{array}
$$

Clearly for any v-proposition Q using original symbols of D, it holds that Q holds in (Φ, σ_0) iff $M \models \pi Q$. A direct consequence is that M is a model of IC_D. It remains to prove that M is a model of $comp_3(P_D)$.

Before continuing with this proof, we want to stress that the complexity of the construction above is in no way an indication that the proposed transformation π is on itself unnecessarily complex or lacks elegance. The increased technicality is only due to the fact that πD can be considered as an untyped meta-program. It is well-known (see e.g. [HL89], [MD92]) that such programs give rise to technical problems with respect to Herbrand semantics. Alternatives would have been to define πD as a typed logic program (as in [Dun93]), or to make its clauses range restricted, using additional range predicates. Both solutions would have reduced the complexity of the proof, but increased the complexity of π itself. This motivates our choice.

That the completed definition of $Noninertial/3$ is satisfied follows straightforwardly: since the expression "p_i holds in $state_M(t_s)$" is equivalent with $M \models Holds(p_i, t_s)$, the completed definition of $Noninertial/3$ is a direct representation of definition 6.2.2.

Finally, we check the completed definition of $Holds/2$. Essentially what must be done is to check all its ground instances $Holds(t_f, t_s) \Leftrightarrow \ldots$. This requires a simple case-analysis depending on the type of t_s. We consider three cases. Take $t_s = s_0$. The completed definition collapses to:

$$Holds(t_f, s_0) \Leftrightarrow Initially(t_f)$$

which is clearly satisfied in M.

Take t_s an empty-state term. The completed definition collapses to:

$$Holds(t_f, t_s) \Leftrightarrow false$$

which is also satisfied in M.

Finally, take $t_s = Result(t_a, t)$. The completed definition collapses to:

$$Holds(t_f, Result(t_a, t)) \leftrightarrow E_1 \vee \ldots E_n$$

with:

$$E_1 = \quad Holds(t_f, t) \wedge \neg Noninertial(t_f, t_a, t)$$
$$E_i(i > 1) = \quad Holds(p_1, t) \wedge \ldots \wedge Holds(p_n, t)$$

such that precisely for each e-proposition

$$t_a \textbf{ causes } t_f \textbf{ if } p_1, \ldots, p_n$$

in D there exists one corresponding disjunct E_i $(i > 1)$ in the formula. Substituting:

$$"t_f \in state_M(Result(t_a, t))"$$

for:

$$Holds(t_f, Result(t_a, t))$$

and substituting:

$$"p_i \text{ holds in } state_M(t)"$$

for:

$$Holds(p_i, t)$$

preserves the truth value of the expression wrt to M. Now observe that by the definition of M, we have

$$t_f \in state_M(Result(t_a, t)) \text{ iff } t_f \in \Phi'(t_a, state_M(t))$$

Hence substituting:

$$t_f \in \Phi'(t_a, state_M(t))$$

for:

$$t_f \in state_M(Result(t_a, t))$$

at the left in the equivalence, preserves again the truth value of the expression. Clearly, we obtain an equivalence as in proposition 6.2.1. This equivalence is satisfied because Φ' is the transition function of a model of D' and D' is e-consistent.

□

Theorem 6.3.2 (completeness) *Let D be e-consistent. For each v-proposition Q, if D entails Q then $\pi D \models \pi Q$.*

Proof Since D is e-consistent, there exists a unique transition function Φ which satisfies the e-propositions of D. As for the soundness, it suffices to prove that for each model M of πD, there exists a model $M_a = (\sigma_0, \Phi)$ of D such that for each v-proposition Q, $M \models \pi Q$ iff Q holds in M_a. Notice that this

immediately implies that all v-propositions of D hold in M_a since M is a model of πQ for each v-proposition Q of D.

M maps each term $Result[a_1;\ldots;a_n,s_0])$ to a domain element, denoted $\tilde{M}(Result[a_1;\ldots;a_n,s_0])$. M_a is defined in the following way: Φ is given; σ_0 is defined as $state_M(\tilde{M}(s_0))$.

We should prove that for each sequence of actions $a_1,\ldots,a_n$:

$$M_a^{a_1;\ldots;a_n} = state_M(\tilde{M}(Result[a_1;\ldots;a_n,s_0]))$$

The proof is by induction on n. For $n=0$, this is trivial. So assume that the theorem holds for $n-1$, $n>0$. We have the following identity:

$$\begin{aligned} M_a^{a_1;\ldots;a_n} &= \Phi(a_n, M_a^{a_1;\ldots;a_{n-1}}) \\ &= \Phi(a_n, state_M(\tilde{M}(Result[a_1;\ldots;a_{n-1},s_0]))) \end{aligned}$$

The second identity follows from the induction hypothesis. Let x be the domain element $\tilde{M}(Result[a_1;\ldots;a_{n-1},s_0])$. It suffices to show that:

$$\begin{aligned} &\Phi(a_n, state_M(\tilde{M}(Result[a_1;\ldots;a_{n-1},s_0]))) = \\ &\qquad state_M((\tilde{M}(Result[a_1;\ldots;a_n,s_0])) \end{aligned}$$

or equivalently:

$$\Phi(a_n, state_M(x)) = state_M(\tilde{M}(Result(a_n,x)))$$

By proposition 6.2.1, we find that $f \in \Phi(a_n, state_M(x))$ iff

- f holds in $state_M(x)$ and f is inertial under a_n in $state_M(x)$, or
- there exists an e-proposition a_n **causes** f **if** $p_1,\ldots,p_m$ such that the fluents $p_1,\ldots,p_m$ hold in $state_M(x)$.

Because M is a model of $Noninertial/3$, the first disjunct corresponds to

$$M \models Holds(f,x) \wedge \neg Noninertial(f,a_n,x)$$

The second disjunct corresponds to the fact that

$$M \models Holds(p_1,x) \wedge \ldots \wedge Holds(p_m,x)$$

for some e-proposition a_n **causes** f **if** $p_1,\ldots,p_m$. Because M is a model of the completed definition of $Holds/2$, we obtain that

$$f \in \Phi(a_n, state_M(x)) \text{ iff } M \models Holds(f, Result(a_n,x))$$

or equivalently

$$f \in state_M(\tilde{M}(Result(a_n,x)))$$

This gives the desired identity.

□

The following example shows that the condition of e-consistency is necessary: π is not complete in general.

Example Consider the following domain description D_2, which uses the fluent *alive* and the action *shoot*.

shoot **causes** *alive*
shoot **causes** $\neg$*alive*

Obviously, D_2 is inconsistent: no transition function Φ can exist which satisfies the two e-propositions. Therefore, each v-proposition is entailed by D_2. πD_2 is given by:

$Holds(F, s_0)$:- $Initially(F)$
$Holds(F, Result(A, S))$:- $Holds(F, S), \neg Noninertial(F, A, S)$
$Holds(alive, Result(shoot, S))$:-
$Noninertial(alive, shoot, S)$:-

This program is consistent. Below, $Result[shoot; \ldots; shoot, s_0]$ is denoted by $shoot^n$. A Herbrand model of πD_2 is given by the set:

$$\{Holds(alive, shoot^n), Noninertial(alive, shoot, shoot^n) \mid n > 0\}$$

In this model, the e-proposition *shoot* **causes** *alive* overrules the contradicting rule *shoot* **causes** $\neg$*alive*. π is not complete since D_2 entails all v-propositions, while πD_2 does not.

When D is inconsistent but e-consistent, then πD is inconsistent too. When D is not e-consistent, then π is incomplete iff πD is consistent. Even in such a case, it is often possible to restore the equivalence between D and πD by extending π as follows. For each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ with f a positive literal, we add the rule:

$$Initiates(a, f, S) \text{ :- } Holds(p_1, S), \ldots, Holds(p_n, S)$$

For each e-proposition a **causes** $\neg f$ **if** $p_1, \ldots, p_n$ with f a positive literal, we add the rule:

$$Terminates(a, f, S) \text{ :- } Holds(p_1, S), \ldots, Holds(p_n, S)$$

In addition, we add the integrity constraint:

$$\forall A, F, S : \neg Initiates(A, F, S) \vee \neg Terminates(A, F, S)$$

Example In πD_2 we have two additional rules:

$$Initiates(shoot, alive, S) \text{ :-}$$

$$Terminates(shoot, alive, S) \text{ :-}$$

It is trivial that the resulting program violates the integrity constraint.

In some interesting situations, this solution does not work. Consider the following example:

Example The domain D_3 is about flipping a (light) switch. There is one action: *switch* and two fluents, *on* and off.

switch **causes** *on* **if** off
switch **causes** off **if** *on*
switch **causes** $\neg$*on* **if** *on*
switch **causes** $\neg off$ **if** off
Initially *on*
Initially $\neg off$

D_3 is not consistent, because $\Phi(switch, \{on, off\})$ is not defined consistently. However, starting from the initial situation in which *on* is true and off is false and applying *switch* consecutively flips the state of *on* and off in such a way that *on* and off are never true in the same state. Hence, from the initial state, the problematic state $\{on, off\}$ can never be reached. For this reason, πD_3 is consistent, even with $Terminating/3$ and $Initiating/3$. Applying the model construction of theorem 3.4.1, we obtain:

$Initially(on)$	
$Holds(on, switch^n)$	for each even n
$Holds(off, switch^n)$	for each odd n
$Noninertial(on, switch, switch^n)$	for each n
$Noninertial(off, switch, switch^n)$	for each n
$Initiates(switch, on, switch^n)$	for each odd n
$Terminates(switch, on, switch^n)$	for each even n
$Initiates(switch, off, switch^n)$	for each even n
$Terminates(switch, off, switch^n)$	for each odd n

For this example, the semantics of D_3 and πD_3 differ. Which semantics is to be preferred? This is a matter of taste, but intuitively we find the domain description D_3 a sensible theory, and the model a sensible model of the theory. By considering D_3 inconsistent, the semantics of $\mathcal{A}$ is to our taste too severe[1].

A final example illustrates why v-propositions are added as integrity constraints and not as program clauses.

Example Take the domain D_4:

a **causes** $\neg f$
f **after** a

[1] Notice that the inconsistency of D_3 can easily be repaired by dropping the fluent off and replacing it everywhere by $\neg$*on*. It is unclear to us whether such a solution exists in general when the semantics of D and πD differ.

Obviously this domain is inconsistent. πD_4 is also inconsistent: indeed the completed definition of $Holds/2$ subsumes

$$Holds(f, Result(a, s_0)) \Leftrightarrow false$$

That contradicts with the integrity constraint $Holds(f, Result(a, s_0))$.

On the other hand, adding $Holds(f, Result(a, s_0))$ as a rule has the effect of adding the disjunct $F = f \wedge T = Result(a, s_0)$ to the completed definition of $Holds$. The resulting theory is consistent and has the model:

$$\{Holds(f, Result(a, s_0)), Noninertial(f, a, t_s) | t_s \in HU(\mathcal{L}_{D_4})\}$$

6.4 Reasoning on incomplete logic programs

Traditionally, incomplete programs have been associated with abduction as procedural paradigm. We argued before that an abductive procedure can also be used for deduction and for proving consistency of a theory. In section 4.6, it was shown how an abductive procedure can be used to prove consistency of a theory. In section 5.6 we argued that an abductive procedure can be used for deduction in an incomplete logic program. Here we illustrate this with SLDNFA.

In a first step the FOL axioms IC_D must be transformed to an incomplete program. The transformation 4.9.1 of IC_D is trivial. A ground atom $Holds(f, Result[a_1; \ldots; a_n, s_0])$ is transformed to:

$$false \text{ :- } \neg Holds(f, Result[a_1; \ldots; a_n, s_0])$$

A ground negative literal $\neg Holds(f, Result[a_1; \ldots; a_n, s_0])$ is transformed to:

$$false \text{ :- } Holds(f, [a_1, ; \ldots; a_n, s_0])$$

Applying this technique on the FOL axioms of the Murder Mystery domain D_1, we obtain an incomplete program P', in which the following rules:

$false$:- $\neg Holds(alive, s_0)$
$false$:- $Holds(alive, [shoot; wait, s_0])$

are substituted for the FOL axioms of πD_1.

An abductive procedure generates explanations for a given observation on the problem domain. Here we can take $\neg false$ as an observation. SLDNFA solves the query $\leftarrow \neg false$ and returns the solution:

$$\Delta_1 = \{Initially(loaded), Initially(alive)\}$$

Not only this gives an explanation for $\neg false$, but proves also that πD_1 is consistent (theorem 4.6.1).

An abductive procedure can also be used for deduction. For example, we want to prove that $\pi D_0 \models Initially(loaded)$ or equivalently that the theory $\pi D + \neg Initially(loaded)$ is inconsistent. To prove that, we add the extra rule :

$$false \text{ :- } Initially(loaded)$$

Now SLDNFA fails finitely on the query $\leftarrow \neg false$. From the first completeness result of SLDNFA, it follows that $\pi D_0 + \neg Initially(loaded)$ is inconsistent. Notice that a completeness result for abduction is used here as a soundness result for deduction.

An abductive procedure allows reasoning under uncertainty. By dropping **Initially** *alive* from the Murder Mystery domain D_1, an incomplete domain description D_1' is obtained. Using $\pi D_1'$, SLDNFA answers the goal $\leftarrow \neg false$ by returning the answer $\Delta_2 = \{\}$. The original solution Δ_1 is still a solution but is not generated. This does not conflict with the completeness result of SLDNFA because $\Delta_2 \subset \Delta_1$.

That we have uncertainty in this domain description becomes obvious when we want to know whether **Initially** *alive* is *possible* according to D_1'. This is done by posing the query $\leftarrow \neg false, Initially(alive)$. SLDNFA proves that **Initially** *alive* is possible by returning Δ_1.

Deduction under uncertainty is possible. Observe that D_1' entails:

$$\textbf{Initially } \neg alive \vee \textbf{Initially } loaded$$

SLDNFA can prove this. This is done by transforming the negation of the disjunction to:

$$false \text{ :- } Initially(loaded)$$

$$false \text{ :- } \neg Initially((alive)$$

After adding these rules to πD_1, SLDNFA fails finitely on $\leftarrow \neg false$. This proves the disjunction.

The above experiments show in the first place that though incomplete/ abductive logic programs are traditionally associated with abduction as procedural paradigm, other procedural paradigms such as deduction and consistency proving are of interest. This illustrates our argument that an *abductive program* is better called an *incomplete program*. In the second place, the experiments show that a suitable abductive procedure can be used to emulate these paradigms.

6.5 The Gelfond & Lifschitz approach

We recall the transformation proposed in [GL92], from $\mathcal{A}$ domain descriptions to extended programs. For any domain description D, $\pi_{GL}D$ is defined as the extended logic program containing the following extended clauses:

- Four inertia rules:

$$Holds(F, Result(A, S)) \leftarrow Holds(F, S), not\ Noninertial(F, A, S) \quad (1')$$
$$\neg Holds(F, Result(A, S)) \leftarrow \neg Holds(F, S), not\ Noninertial(F, A, S) \quad (2')$$
$$Holds(F, S) \leftarrow Holds(F, Result(A, S)), not\ Noninertial(F, A, S) \quad (3')$$
$$\neg Holds(F, S) \leftarrow \neg Holds(F, Result(A, S)), not\ Noninertial(F, A, S) \quad (4')$$

- Each v-proposition f **after** $a_1; \ldots; a_n$, is transformed into:

$$Holds(f, Result[a_1; \ldots; a_n, s_0]) \quad (5')$$

  Recall that $Holds(\neg f, \ldots)$ denotes $\neg Holds(f, \ldots)$.

- Each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ is translated into 2n+2 rules. Below, $\overline{Holds(f, S)}$ denotes the complement of $Holds(f, S)$ with respect to $\neg$.

$$Holds(f, Result(a, S)) \leftarrow Holds(p_1, S), \ldots, Holds(p_n, S) \quad (6')$$
$$Noninertial(|f|, a, S) \leftarrow not\ \overline{Holds(p_1, S)}, \ldots, not\ \overline{Holds(p_n, S)} \quad (7')$$

  For each i, $1 \leq i \leq n$:

$$\overline{Holds(p_i, S)} \leftarrow \overline{Holds(f, S)}, Holds(f, Result(a, S)) \quad (8')$$
$$\overline{Holds(p_i, S)} \leftarrow \overline{Holds(f, Result(a, S))}, Holds(p_1, S), \ldots, Holds(p_{i-1}, S), Holds(p_{i+1}, S), \ldots, Holds(p_n, S) \quad (9')$$

[GL92] gives the intuition behind the translation and gives a soundness theorem for all domain descriptions D provided D does not contain *similar* e-propositions, i.e. e-propositions which only differ by the preconditions. A comparison of $\pi_{GL}D$ with πD is of interest. Observe that if the two negations *not* and $\neg$ in extended programs are mapped both on "$\neg$" in incomplete programs, we find clauses or formulas in πD corresponding to (1′), (5′), (6′) (if f is a positive literal) and (7′), while (2′), (3′), (4′), (8′) and (9′) lack in πD.

A striking fact is that $\pi_{GL}D$ contains four inertia rules instead of one in πD. (1′) and (2′) are *forward persistence rules* for respectively positive and negative fluents. (3′) and (4′) are *backward persistence rules* for again positive and negative fluents. Clearly (2′), (3′) and (4′) are natural rules, which are expected to hold in any correct formalisation. Therefore, they must be subsumed by πD, otherwise π could never be sound and complete. As a matter of fact, it is straightforward to prove that for each of the extended rules in $\pi_{GL}D$, the corresponding clause is subsumed by $comp_3(P_D)$, where P_D is the logic program part of πD. For example, notice that from the classical logic point of view the rules (1′) and (4′) are equivalent and so are the rules (2′) and (3′). This immediately gives that $comp_3(P_D)$ subsumes (4′). Clauses corresponding to (2′) and (3′) can be derived from the completed definition

(6.6) of $Holds/2$ in P_D. Substitute $Result(A,S)$ for T. After simplification one obtains:

$$\forall F, A, S : Holds(F, Result(A,S)) \Leftrightarrow E_1 \vee \ldots \vee E_n$$

where E_1 is of the form:

$$Holds(F,S) \wedge \neg Noninertial(F,A,S)$$

and for each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ with f is a positive literal, there is an E_i of the form:

$$F = f \wedge A = a \wedge Holds(p_1,S) \wedge \ldots \wedge Holds(p_n,S)$$

Now, it is easy to see that $comp_3(P_D)$ satisfies:

$$\forall F, A, S : F = f \wedge A = a \wedge Holds(p_1,S) \wedge \ldots \wedge Holds(p_n,S) \Rightarrow Noninertial(F,A,S)$$

By dropping $\neg Noninertial(F,A,S)$ from the first disjunct and substituting $Noninertial(F,A,S)$ for the other disjuncts, we find:

$$\forall F, A, S : Holds(F, Result(A,S)) \Rightarrow Holds(F,S) \vee Noninertial(F,A,S)$$

Simple rewriting gives formulas corresponding to $(2')$ and $(3')$.

A shortcoming of π_{GL} is its incompleteness. [GL92] gives the following example D_5:

a **causes** f **if** f
f **after** a

Clearly D_5 entails **Initially** f. However, $Initially(f)$ is not entailed by $\pi_{GL}D_5$. On the other hand, notice that D_5 is e-consistent. Therefore, $Initially(f)$ is implied by πD_5 and can be proven by SLDNFA.

Another problem of π_{GL} shows up when $\mathcal{A}$ is extended to allow predicates. Consider the following rule:

$$Pick(X,Obj) \textbf{ causes } thief(X) \textbf{ if } owner(Y,Obj), X \neq Y$$

which says that X becomes a thief if he picks an object Obj which he does not own. The translation to incomplete programs does not require any modification. π produces:

$$Holds(thief(X), Result(Pick(X,Obj),S)) \text{ :- } Holds(owner(Y,Obj),S), X \neq Y$$
$$Noninertial(thief(X), Pick(X,Obj),S) \text{ :- } Holds(owner(Y,Obj),S), X \neq Y$$

For π_{GL}, there are problems with the rules of type $(8')$:

$$Holds(owner(Y,Obj),S) \leftarrow \neg Holds(thief(X),S), Holds(thief(X),Result(Pick(X,Obj),S))$$
$$X \neq Y \qquad \leftarrow \neg Holds(thief(X),S), Holds(thief(X),Result(Pick(X,Obj),S))$$

These rules say that when X becomes thief by picking something in situation S, then each Y is owner at situation S and no Y is equal to X. This is a contradiction. The problem is that Y should not be universally but existentially quantified. The following formulas are subsumed by πD but are not extended clauses:

$$\forall Obj,S,X : \exists Y : Holds(owner(Y,Obj),S) \leftarrow \neg Holds(thief(X),S), Holds(thief(X),Result(Pick(X,Obj),S))$$
$$\forall Obj,S,X : \exists Y : X \neq Y \qquad \leftarrow \neg Holds(thief(X),S), Holds(thief(X),Result(Pick(X,Obj),S))$$

The translation π to incomplete programs performs better than the translation π_{GL} to extended programs. π_{GL} creates a higher number of rules, is incomplete, suffers from problems with similar e-propositions and is not directly extendible to the predicate case. The incomplete program approach seems more understandable because only one negation occurs, is sound even with similar e-propositions, is complete for all reasonable domain descriptions and applies without modification for the predicate case.

6.6 Dung's approach

In [Dun93] Dung presents another translation from $\mathcal{A}$ domains to a logic program formalism, which is quite similar to ours in a number of aspects. On the syntactical level, the most important difference with our approach is the symmetrical treatment of fluent symbols f and their negation $\neg f$. The translation $\pi_{Du}D$ contains our frame axiom (6.2), and contains for each e-proposition a **causes** f **if** $p_1,\ldots,p_n$ (f positive or negative) the following rules:

$$Holds(f,Result(a,S)) \text{ :- } Holds(p_1,S),\ldots,Holds(p_n,S)$$
$$Noninertial(f,a,S) \text{ :- } Holds(p_1,S),\ldots,Holds(p_n,S)$$
$$Noninertial(f^*,a,S) \text{ :- } Holds(p_1,S),\ldots,Holds(p_n,S)$$

Here f^* denotes the complement of f. Contrary to our approach, here fluent literals like $\neg f$ appear within $Holds/2$. Each v-proposition f **after** $a_1;\ldots;a_n$ is transformed to the denial:

$$\leftarrow Holds(f^*,Result(a_1;\ldots;a_n,s_0))$$

In addition, for each fluent symbol f the following constraints are added:

$\leftarrow Holds(f,S), Holds(\neg f,S)$
$Holds(\neg f,s_0) \Leftrightarrow \neg Holds(f,s_0)$

These additional constraints are necessary due to the symmetrical treatment of a fluent f and its negation $\neg f$. This redundancy leads to substantially more rules than in our transformation.

The semantics of $\pi_{Du}D$ is defined via a domain dependent variant of the completion semantics. It contains FEQ and the normal completed definition of $Noninertial/3$, but a specialised version of the completed definition of $Holds/2$. $\pi_{Du}D$ does not contain rules with $Holds(F,s_0)$ in the head. As a consequence the standard completion would imply that

$$\forall F : \neg Holds(F,s_0)$$

[Dun93] avoids this by the following alternative:

$\forall F,A,T : Holds(F,Result(A,S)) \leftrightarrow E_1 \vee \ldots \vee E_n$
with:
$E_1 = \quad Holds(F,S) \wedge \neg Noninertial(F,A,S)$
$E_i(i>1) = \quad F = f \wedge A = a \wedge Holds(p_1,S) \wedge \ldots \wedge Holds(p_n,S)$

such that precisely for each e-proposition a **causes** f **if** $p_1,\ldots,p_n$ (f positive or negative), there exists one disjunct E_i ($i>1$) in the completed definition. This formula says nothing about $Holds(F,s_0)$, and therefore, we get a similar semantics as in our approach, but without $Initially/1$. Dung extends $\mathcal{A}$ to the predicate case and applies a partial deduction procedure in order to generate abductive solutions for queries on $\pi_{Du}D$. He gives an application for checking the satisfiability of a database update with respect to a deductive database with integrity constraints.

A disputable statement in [Dun93] is related to the following example D_6 (a syntactical simplification of the example in theorem (8) in [Dun93]):

Initially f
a **causes** $\neg f$ **if** g

Dung observes correctly that the Gelfond and Lifschitz transformation $\pi_{GL}D_6$ has two answer sets:

$Z_1 = \{Holds(f,s_0)\} \cup \{Noninertial(f,a,a^n) \mid n \in \mathbb{N}\}$
$Z_2 = \{Holds(f,a^n), \neg Holds(g,a^n) \mid n \in \mathbb{N}\}$

In Z_1, $Holds(f,a^n)$ ($n>0$) and $Holds(g,a^n)$ ($n \geq 0$) are unknown since neither the atom nor its negation appears in Z_1. Z_2 corresponds to a two-valued model, obtained by having f initially true and g false, a situation which is preserved when applying a. Then Dung argues that "*it is obvious that only the first solution captures the intended semantics of D_6, for if we don't know anything about g, it*

is impossible to say anything about the outcome of a". Remarkable now is that Z_2 corresponds to a model of Dung's $\pi_{Du}D_6$.

Clearly, Dung views Z_2 as a *knowledge state* model. As we defined in section 2.2, a knowledge state model describes what atoms are known to be true, what atoms are known to be false and what atoms are unknown. Under this view, indeed, Z_2 is incorrect, since we do not know that $Holds(g, s_0)$ is false. However, under the alternative interpretation of a model as a *possible state*, there is no problem with this model: it just represents the possible state that g is initially false and f remains true after applying a.

In a possible state semantics, the fact of having incomplete knowledge on g in the initial state, is reflected by the fact that there are models in which g is initially true and others in which g is initially false. In a model in which g is initially true, f is necessarily terminated after applying a. When g is initially false, as in Z_2, f remains true. It turns out that the two transformations π and π_{Du} behave correct under this view. Note that D_6 has two models, one with initial situation $\{f\}$ and another with initial situation $\{f, g\}$. These models correspond to models of πD_6 and, as Dung perhaps has not noticed, with models of $\pi_{Du}D_6$. Both models are erroneous as knowledge state models. For $\pi_{GL}D_6$, the second model, corresponding to the possible state with initial situation $\{f, g\}$, is weakened to the answer set Z_1.

Of the two type of semantics, the possible state view is definitely the richest one. Indeed, consider the following formulas:

$$\neg Holds(g, s_0) \Rightarrow [Holds(f, s_0) \Rightarrow Holds(f, Result(a, s_0))]$$

$$\neg Holds(g, s_0) \Rightarrow [\neg Holds(f, s_0) \Rightarrow \neg Holds(f, Result(a, s_0))]$$

$$Holds(g, s_0) \Rightarrow \neg Holds(f, Result(a, s_0))$$

$$Holds(f, Result(a, s_0)) \vee Holds(g, Result(a, s_0))$$

Surely one will agree that they are intuitively right. As a matter of fact, they are true in all models of πD_6 and $\pi_{Du}D_6$ and hence, they are implied by these theories. SLDNFA can prove each of them in πD_6 (likely Dung's procedure can do so for $\pi_{Du}D_6$). On the other hand, they are not true in the answer set Z_1, and hence $\pi_{GL}D_6$ does not imply them. The problem with the incompleteness of π_{GL} is not due to models like Z_2 but due to models like Z_1.

6.7 Discussion

[Rei92] formalises database evolution using situation calculus theories in First Order Logic. The completion of a program πD shows a strong relationship with these theories. [Rei92] replaces $Result/2$ by $do/2$. Instead of using the meta predicate $Holds/2$, each fluent predicate is added one additional argument; i.e. an atom

$Holds(p(x), t)$ is contracted to the atom $p(x, t)$. As a consequence the law of inertia has to be stated for each fluent.

The formalism in [Rei92] can deal with additional features such as necessary preconditions for actions, with queries quantified over all times, with defined predicates and with indeterminate actions. An example of a formula which occurs in his approach is:

$$\forall St, C, A, S : Poss(A, S) \rightarrow (enrolled(St, C, do(A, S)) \Leftrightarrow A = register(St, C) \vee enrolled(St, C, S) \wedge A \neq drop(A, C))$$

The rule says that when action A may be executed in situation S ($Poss(A, S)$), then student St is enrolled in course C at time $do(A, S)$ iff A is an action of registering St in C or, St was enrolled at S and A is not an action of dropping St from the course C. If we forget about $Poss(A, S)$, and introduce $Result$ and $Hold$, we find:

$$\forall St, C, A, S : Holds(enrolled(St, C), Result(A, S)) \Leftrightarrow A = register(St, C) \vee Holds(enrolled(St, C), S) \wedge A \neq drop(St, C)$$

Similar formulas are subsumed by πD. The first disjunct corresponds to a rule initiating $enrolled(St, C)$ by $enregister(St, C)$. The second disjunct corresponds to the law of inertia, with $\neg Noninertial(enrolled(St, C), A, S)$ replaced by its definition: $drop(St, C)$ is the only action which terminates $enrolled(St, C)$[2].

In the past, another approach has been explored for temporal reasoning, based on event calculus [KS86]. [Esh88] and [Sha89] have simplified event calculus and have extended it with abduction for the purpose of planning. [Sha89] extended event calculus to deal with necessary preconditions of actions. [Mis91a] implemented a planning system based on this formalism. Other work has been done to extend event calculus with continuous actions [Sha90] and time granularity [Eva90], [MMCR92]. Recently [DMB92] applied abductive event calculus to solve a number of benchmark problems in temporal reasoning, such as the Murder Mystery, the Stolen Car problem, the Walking Turkey Shooting problem and the Russian Turkey Shooting problem. The latter problem contains an indeterminate action. Situation and event calculus seem two non-equivalent ways of representing time and action. $\mathcal{A}$ domain descriptions cannot (easily) be translated to event calculus, because $\mathcal{A}$ assumes a situation calculus philosophy. A deep analysis of situation versus event calculus is beyond the scope of the chapter.

It turns out that the technique used in [DMB92] to represent indeterminate actions can easily be translated to situation calculus. The Russian Turkey Shooting problem is a variant of the Yale Turkey Shooting problem in which one

[2]Here the version of $Noninertial/3$ is needed which contains only rules for terminating effects of actions.

additional action *spinning* of spinning the gun's chamber occurs. The effect is that the gun is possibly unloaded. Below we allow e-propositions of the form a **possibly causes** f **if** $p_1, \ldots, p_n$. The problem is formalised as follows:

Initially *alive*
Initially *loaded*
load **causes** *loaded*
shoot **causes** $\neg$*alive* **if** *loaded*
shoot **causes** $\neg$*loaded*
spinning **possibly causes** $\neg$*loaded*

The semantics of $\mathcal{A}$ can easily be adapted. While in $\mathcal{A}$, a successor state is completely determined by the action and the previous state, this is not the case with indeterminate actions. Therefore, in the extended version the transition function should be replaced by a transition relation. In the corresponding incomplete program, the indeterminism is captured by introducing an undefined $Good\_luck/2$ predicate:

$$Noninertial(loaded, spinning, S) \text{ :- } Good\_luck(spinning, S)$$

The above clause has the effect that the rule of inertia is disabled for *loaded* iff good luck occurs at the spinning action in state S. In general, for each clause

$$a \textbf{ possibly causes } f \textbf{ if } p_1, \ldots, p_n$$

the following rule must be introduced:

$$Noninertial(|f|, a, S) \text{ :- } Holds(p_1, S), \ldots, Holds(p_n, S), Good\_luck(a, S)$$

For a positive f, in addition the following rule is added:

$$Holds(f, Result(a, S)) \text{ :- } Holds(p_1, S), \ldots, Holds(p_m, S), Good\_luck(a, S)$$

6.8 Summary

We presented a sound and complete transformation π from $\mathcal{A}$ domains to incomplete logic programs with FOL axioms. We have illustrated the use of SLDNFA for abductive and deductive reasoning under uncertainty and satisfiability proving. The transformation of Gelfond and Lifschitz is more complex, is not complete, is only sound for domains without e-similar actions and cannot be extended to the predicate case (at least not without imposing other syntactic constraints). Moreover, no reasoning procedure is currently described for the resulting programs. Dung's approach is in many aspects similar to ours and provides a reasoning procedure, but is still more complex than ours, has the disadvantage of relying on a

special purpose logic and does not show the application of the reasoning procedure for other forms of reasoning than abduction.

We have investigated also a number of typical temporal reasoning issues. Although in πD only forward persistence axioms are contained, the completion of πD subsumes backward persistence axioms. We have also shown how to extend $\mathcal{A}$ with indeterminate actions.

From a more general perspective, this work can be viewed as a -successful- experiment in the declarative representation of and diverse forms of automated reasoning on incomplete knowledge using incomplete logic programming and an abductive procedure.

Chapter 7

Temporal reasoning in Incomplete Event Calculus

7.1 Introduction

Event Calculus was developed by Kowalski and Sergot in [KS86] to cope with representational and computational problems of situation calculus in the context of declarative database updates. An event calculus represents a temporal domain by describing events and how they affect the state of the world. Originally, event calculus was based on the complete logic programming formalism. In [Esh88], Eshghi showed that abduction in event calculus could be used to solve planning problems. This approach was further explored by Shanahan [Sha89] and Missiaen [MBD92]. In a planning problem, a set of events must be found which transforms a given initial state into a final goal state. By defining the predicates which describe the events, their associated actions and their order as abducible, an abductive procedure solves a query representing the goal state by returning a description of a set of events which constitutes a *plan*.

The above studies have motivated much of the work presented in this thesis. The development of SLDNFA (chapter 5) was partially motivated by the problems of existing procedures in the context of planning with abductive event calculus. The success of abductive logic programs for representing planning problems was one of the motivations for our investigation of the declarative properties of abductive/incomplete logic programming, the result of which is presented in chapter 4.

So far, most work in event calculus focussed mainly on the procedural semantics. E.g. [KS86] explains the concepts of event calculus using procedural concepts such as default reasoning and negation as failure; [Esh88], [Sha89], [Mis91a] introduce abduction without investigating the declarative semantics of abductive event calculus. In this chapter we show that incomplete event calculus is an elegant and

general *declarative* paradigm for representing temporal domains. An event calculus has one or more models which represent the possible states of the intended temporal domain. We can investigate the declarative reading of event calculus under completion semantics by reading the completed definitions or under justification semantics, by reading an event calculus as a set of constructive definitions.

This declarative view allows to address a number of important issues. One issue is the representation of incomplete knowledge. Remarkably, although the use of an abductive procedure in event calculus is well-known and although abduction is clearly a procedural paradigm for reasoning on incomplete knowledge, so far the role of abductive/incomplete logic programming as a declarative formalism for representing incomplete knowledge has not been recognised. We show how incomplete event calculus gracefully allows to represent diverse forms of incomplete knowledge:

- on the initial state,
- on the order of a known set of events,
- on the events and their order
- on the effect of (indeterminate) events

Such incomplete knowledge can be represented in the same way as in chapter 6: in a given problem domain, the primitive relations on which incomplete knowledge exist, are identified and are represented by undefined predicates. Partial knowledge about them is represented by integrity constraints. The technique is elegant and generally applicable. In none of the examples, the clauses representing the inertia laws need to be changed.

A second important issue is the nature of time. In [KS86] it is argued that one of the advantages of (complete) event calculus over situation calculus is that in a complete event calculus, an incompletely known time precedence relation $\ll$ on the events can correctly be represented by a partial order. Analogously, in previous approaches in planning, the underlying theory of time is the theory of partial order. We show that these approaches are too weak and may lead to bad answers. We argue that a correct complete or incomplete event calculus should entail that $\ll$ is a linear or total order. For a complete event calculus, time points should be linearly ordered. For an incomplete event calculus with undefined $\ll$, the condition can be enforced by adding the theory of linear time as a set of integrity constraints. That the time precedence relation in complete event calculus should always be a linear order implies that it cannot represent incomplete knowledge on $\ll$. However, this does not restrict the expressive power of event calculus: when one has incomplete knowledge on $\ll$, one can make $\ll$ undefined, and add partial knowledge on it as a set of integrity constraints.

In a planning problem, the set of events and their order is the subject of the search and hence a fortiori we have incomplete knowledge on $\ll$. In principle, SLDNFA (and any other sound abductive procedure which can deal with non-ground abducible atoms) is able to solve planning problems. However, the presence

of the theory of linear order causes important efficiency problems. Not only the manipulation of this theory is computationally expensive, also SLDNFA will only generate linear *plans*, i.e. linearly ordered sets of events. In general, *partial plans*, which leave open the order of independent events whose order is irrelevant for the final state, are preferable over linear plans. The number of plans computed by a linear planner compared to a partial planner, is exponential with the number of independent pairs of events. In section 7.3, we extend SLDNFA with a constraint solver which checks the satisfiability of the abduced time precedence facts against the theory of linear order and we prove the soundness and a completeness result. The resulting procedure offers not only a correct and more efficient treatment of time, it also generates correct partial plans. In section 7.6, we demonstrate SLDNFA-LO for some planning examples.

The system is not only useful in the context of planning but can be used for temporal reasoning in general. To illustrate this, we present solutions for a number of well-known temporal reasoning problems. These problems are considered as important benchmarks for temporal reasoning formalisms [San91]. On the declarative level, many of these problems involve the representation of incomplete knowledge either on the initial state or on the events, their order or their effects. On the procedural level, they involve complex reasoning such as prediction, ambiguous prediction, postdiction and ambiguous postdiction. We show how SLDNFA-LO is useful to perform deduction, abduction and satisfiability proving on these theories.

The chapter is structured as follows. In section 7.2 a simplified version of event calculus is introduced. Examples illustrate the representation of incomplete knowledge. We investigate the role of time as a linear order. In section 7.3, SLDNFA is extended with a constraint module for the theory of linear order. In section 7.4, several important issues are considered: the notions of event versus action, necessary pre-conditions for events and context dependent effects of events, indeterminate events, the ramification problem in event calculus, actions with duration, concurrent actions. In section 7.5, we point to some declarative singularities in the completion and justification semantics of event calculus. In section 7.7 a discussion of future extensions and related work is given.

This chapter is a serious extension of [DMB92].

7.2 A theory on time, state, action and change

In this section we will introduce a basic version of event calculus. This version is analogous to the one used in [Sha90] and [Mis91b, Mis91a] but attaches a different interpretation to the predicates. In contrast to other approaches, the ontological primitive in the version below is the *time point* instead of the *event*. This choice is motivated by our intention to investigate the laws of time.

An event calculus consist of a possibly incomplete logic program and a set of integrity constraints. It describes time, time dependent properties, events and

change caused by the events. The constants of the language have domain dependent objects as intended interpretation, like *john*, *block_A*, or they denote time points, like *t_1*, etc.. Functors may denote functions on the problem domain, e.g. $father\_of(john)$, but also the time dependent relations of the problem domain, e.g. the term $on(block\_A, block\_B$ represents the relation between two blocks in the problem domain. In the sequel, the latter functors will be called *fluents*. One subset of predicates represents the time independent properties of the problem domain. In addition there are a number of generic, domain independent predicates which are introduced to describe how events affect the world. They are listed below together with their intended interpretation.

- $Time(T)$: T is a time point. $Time/1$ is a type predicate, representing the time points.
- $T_1 \ll T_2$: time point T_1 precedes time point T_2.
- $Act(T, A)$: on time T, an event A occurs. Events have no duration.
- $Holds\_at(P, T)$: the fluent P holds at time T.
- $Clipped(E, P, T)$: the fluent P becomes false in the half open interval $[E, T[$.
- $Initiates(T, P)$: at time T there is an event with an initiating effect on the fluent P.
- $Terminates(T, P)$: at time T there is an event with a terminating effect on the fluent P.

A few remarks are in order. First, predicates like $Holds\_at/2$, $Clipped/3$, $Initiates/2$, $Terminates/2$ have fluents as argument and hence can be considered as simple "meta-predicates". Keep in mind however that their arguments should be simple fluent terms and not composed formulas.

Second, we should be careful about the meaning of *an event having an initiating or terminating effect on some fluent* P. By default, this means that P is true after an initiating event and false after a terminating event. This does not exclude that P was true before the initiating event or false before the terminating event. For example to shoot on a turkey has a terminating effect on the *live* of the turkey, disregarding whether the turkey was alive or dead before.

Third and most important is a comment on the nature of time in event calculus. Is time a finite or countable discrete set of time points, or is it isomorphic with the rational or real numbers? The event calculus has models with discrete finite time and other models with rational or real time. Although people tend to view time as a continuous and dense set, isomorphic with the real numbers, commonsense reasoning often does not rely on the laws of a continuous time and is correct also in a discrete finite time. A discrete finite time can be interpreted as a set of

relevant time points, on which events occur or on which observations are made. In section 7.7 we discuss how to obtain an instance of event calculus in which time is isomorphic with the real numbers.

In our version of event calculus, the law of inertia, sometimes called the persistence axiom or the frame axiom, is formulated as follows:

$$\begin{aligned}
&Holds_at(P,T) \text{ :- } Time(E), E \ll T, Initiates(E,P),\\
&\qquad\qquad \neg Clipped(E,P,T)\\
&Clipped(E,P,T) \text{ :- } Time(C), Terminates(C,P), In(C,E,T)\\
&In(C,C,T) \text{ :-}\\
&In(C,E,T) \text{ :- } E \ll C, C \ll T
\end{aligned}$$

These laws state that a fluent P is true at some time T if there is a strictly earlier time E on which there is an initiating event for P and such that P is not clipped: there is no terminating event in the interval $[E,T[$. $In(C,E,T)$ has intended interpretation that C occurs in $[E,T[$. One easily verifies that the state of the world on the moment that an event occurs is not affected by the event: initiating and terminating effects on time T do not affect $Holds\_at(P,T)$. This avoids problems in case an event has a terminating effect on its preconditions.

The initiating and terminating effects of actions are described by the definitions of the predicates $Initiates/2$ and $Terminates/2$. These will in general consist of rules of the form:

$$\begin{aligned}
&Initiates(E,p) \text{ :- } Act(E,a), Holds_at(p_1,E), \ldots, Holds_at(p_k,E),\\
&\qquad\qquad \neg Holds_at(p_{k+1},E), \ldots, \neg Holds_at(p_l,E)\\
&Terminates(E,p) \text{ :- } Act(E,a), Holds_at(p_1,E), \ldots, Holds_at(p_k,E),\\
&\qquad\qquad \neg Holds_at(p_{k+1},E), \ldots, \neg Holds_at(p_l,E)
\end{aligned}$$

These clauses define that an action a, occurring at a time E on which the fluents $p_1, \ldots, p_k$ hold and the fluents $p_{k+1}, \ldots, p_l$ do not hold, has the effect of initiating or terminating the fluent p.

A general way to represent an initial situation is by introducing a time point *start* and a predicate $Initially/1$. The definition of $Initiates/2$ should contain the following rule:

$$Initiates(start,P) \text{ :- } Initially(P)$$

In addition, the program should imply the following formulas:

$$\begin{aligned}
&Time(start)\\
&Time(E) \rightarrow start = E \vee start \ll E
\end{aligned}$$

In a complete event calculus these formulas should be entailed by the program. When $Time/1$ and/or $\ll$ are undefined, they may be added as a set of explicit integrity constraints.

So far we obtained definitions for the predicates $Holds\_at/2$, $Clipped/3$, $In/3$, $Initiates/2$ and $Terminates/2$. In addition, the other predicates $Time/1$, $\ll$, $Initially/1$ and $Act/2$ may have problem specific definitions, depending on whether we have complete information on them or not. An example in which we have complete information on all of them is the Yale Turkey Shooting problem. Recall the problem specification from chapter 6: *a turkey is alive initially; there are successively events of loading a gun, waiting and shooting.* In event calculus the problem is represented as follows:

Domain independent clauses:

$Holds\_at(P,T)$:- $Time(E), E \ll T, Initiates(E,P),$
$\neg Clipped(E,P,T)$

$Clipped(E,P,T)$:- $Time(C), Terminates(C,P), In(C,E,T)$

$In(C,C,T)$:-

$In(C,E,T)$:- $E \ll C, C \ll T$

$Initiates(start,P)$:- $Initially(P)$

Domain dependent clauses:

$Initiates(T,loaded)$:- $Act(T,loading)$

$Terminates(T,loaded)$:- $Act(T,shooting)$

$Terminates(T,alive)$:- $Act(T,shooting), Holds\_at(loaded,T)$

In addition, for each of the following problem specific atoms, one atomic clause is added:

$Initially(Alive)$

$Time(start), Time(e_1), Time(e_2), Time(e_3), Time(t_{end})$

$Act(e_1,loading), Act(e_2,waiting), Act(e_3,shooting)$

10 atomic clauses for (or a definition of $\ll$ as) the transitive closure of the following time precedence facts are needed:

$start \ll e_1 \ll e_2 \ll e_3 \ll t_{end}$

The result is a complete logic program. The semantics of the program is given by the justification semantics. However, remember from chapter 4 that direct justification semantics and 3-valued completion semantics gives always a safe approximation of justification semantics and that the completed definitions of predicates hold under justification semantics. In most examples which occur below, the completion semantics will be sufficiently precise, and SLDNF or SLDNFA will be able to prove relevant properties. We refer to section 7.5 for a discussion when the completion semantics is not sufficiently precise.

Under 3-valued completion semantics, the program correctly implies that the turkey is dead at t_{end}. Moreover, SLDNF can prove this: SLDNF succeeds on the goal $\leftarrow \neg Holds\_at(alive, t_{end})$.

Many classical temporal reasoning problems involve reasoning on incomplete knowledge. As in chapters 4 and 6, we advocate the use of incomplete logic programming to represent such domains. The technique is based on the following simple principle: given a specification, we identify the primitive predicates on which

there is incomplete information or for which a complete definition cannot be given. These predicates appear undefined in the resulting theory. Partial knowledge on them is represented via integrity constraints.

A well-known example given also in chapter 6, is the Murder Mystery [Bak89]: *initially the turkey is alive; there is a shooting and a waiting event; then the turkey is dead.* In this problem there is full knowledge on the events and their order but there is incomplete information on the initial situation. Hence, $Initially/1$ is the only undefined predicate. The domain independent information and the general domain knowledge is as in the YTS solution. For each of the following problem specific atoms, one atomic clause is added:

$Time(start), Time(e_1), Time(e_2), Time(t_{end})$
$Act(e_1, shooting), Act(e_2, waiting)$

In addition, 6 atomic clauses for (or a definition of $\ll$ as) the transitive closure of the following time precedence facts are needed:

$start \ll e_1 \ll e_2 \ll t_{end}$

Two integrity constraints:

$Initially(alive), \neg Holds\_at(alive, t_{end})$

This is a correct representation of the problem. $Initially(loaded)$ is implied by the completion. SLDNFA can prove this. After transforming the integrity constraints to:

$false \leftarrow \neg Initially(alive)$
$false \leftarrow Holds\_at(alive, t_{end})$

SLDNFA finitely fails on the goal $\leftarrow \neg Initially(loaded), \neg false$.

In planning problems, the goal is to find a set of events which transform a given initial state to a given goal state. From the pure declarative point of view, a theory describing the planning domain describes the effects of actions and represents initial and goal state, but there is incomplete knowledge on the events and their order. Hence, in such a theory the predicates $Time/1$, $Act/2$ and $\ll$ should appear undefined. However, there is a snake under the grass. Intuitively, it is clear that in any possible "world", time is a linear (or total) order. To have incomplete knowledge about the order of two time points t_1, t_2 means that there are possible worlds in which they are identical, other in which t_1 occurs before t_2 and yet other in which t_2 occurs before t_1. Hence, a correct event calculus should logically imply the theory of linear order. One way to enforce this when $\ll$ is undefined is to add the theory of linear time as a set of integrity constraints:

$X \ll Y \rightarrow Time(X) \wedge Time(Y)$	$\ll$ is well-typed
$X \ll Y \wedge Y \ll Z \rightarrow X \ll Z$	transitivity
$\leftarrow X \ll Y, Y \ll X$	asymmetry, irreflexivity
$Time(X) \wedge Time(Y) \rightarrow X \ll Y \vee X = Y \vee Y \ll X$	linearity

Our solution differs from existing proposals. [Esh88] adds the theory of partial order. [Mis91a] does not add this theory explicitly but incorporates in his abductive procedure a constraint solver for the theory of partial order. The issue of the linearity of time appears also in the context of complete event calculus. In [KS86], it is argued that one of the advantages of (complete) event calculus over situation calculus is that the time precedence of events can be a partial order representing an incompletely known order of events. However, a complete event calculus which defines $\ll$ as a partial order has only models in which the axiom of linearity is violated. This problem on the level of the declarative semantics causes problems on the procedural level when unordered events are dependent, i.e. when they affect each others preconditions. An example shows the problem: *initially the light is off; at two different times* e_1, e_2, *a light switch is flipped; the order of* e_1, e_2 *is unknown.* We should be able to infer that the light is off at the final state. The logic is specified as follows:

Domain dependent clauses:
$Initiates(E, on)$:- $Act(E, flip\_switch), \neg Holds\_at(on, E)$
$Terminates(E, on)$:- $Act(E, flip\_switch), Holds\_at(on, E)$
Problem specific information: for each atom below an atomic clause is added:
$Time(start), Time(e_1), Time(e_2), Time(t_{end})$
$Act(e_1, flip\_switch), Act(e_2, flip\_switch)$
$start \ll e_1, start \ll e_2, e_1 \ll t_{end}, e_2 \ll t_{end}$

This is a complete event calculus, which defines a partial order on time. Note that $Initially/1$ has an empty definition, which implies that the light is off initially.

This program entails that the light is on at t_{end}. SLDNF answers yes on the goal $\leftarrow Holds\_at(on, t_{end})$. This can be seen as follows. Consider the following instance of the law of inertia:

$$Holds\_at(on, t_{end}) \text{ :- } Time(E), E \ll t_{end}, Initiates(E, on), \neg Clipped(E, on, t_{end})$$

It has two instances with true body for $E = e_1$ and $E = e_2$. We sketch the proof for $Initiates(e_1, on)$ and $\neg Clipped(e_1, on, t_{end})$ in the first instance. At *start*, the light is off. The only event which satisfies $E \ll e_1$ is *start* itself. Since *start* does not initiate *on*, *on* is provably not initiated between *start* and e_1. Hence *on* is false when e_1 occurs. So, e_1 provably initiates *on*. To prove $\neg Clipped(e_1, on, t_{end})$, observe that the only event E which satisfies $In(E, e_1, t_{end})$ is e_1 itself. e_1 does not terminate *on*. Hence, $Clipped(e_1, on, t_{end})$ is provably false.

The fault in the proof is clearly the assumption that e_2 neither satisfies $e_2 \ll e_1$ in the proof of $Initiates(e_1, on)$ nor $In(e_2, e_1, t_{end})$ in the proof of $\neg Clipped(e_1, on, t_{end})$. The problem is not due to SLDNF: the program implies $\neg e_2 \ll e_1 \wedge e_1 \neq e_2 \wedge \neg e_1 \ll e_2$ which violates of the law of linearity. From this formula, the two assumptions can correctly be proven. In general, any complete logic program which

represents $\ll$ by a non-linear partial order implies at least one counter-intuitive formula: if e_1, e_2 are unrelated distinct time points, then $\neg e_1 \ll e_2 \wedge e_1 \neq e_2 \wedge \neg e_2 \ll e_1$ is entailed. This is a violation of the law of linearity.

The problem appears also in the context of incomplete event calculus. Assume that in the light switch problem the definition of $\ll$ is dropped and the $\ll$ atoms are added as integrity constraints. If the axiom of linear order is not added explicitly, there exists a model of the program in which $Holds\_at(on, t_{end})$ is true and e_1 and e_2 are unordered. SLDNFA solves the goal $\leftarrow Holds\_at(on, t_{end}), \neg false$ by returning an answer in which e_1 and e_2 are unrelated in time. This type of problem is similar to the problems reported in [Mis91a] on the planning approaches in [Esh88], [Sha89], [Mis91a].

By adding the theory of linear order, SLDNFA and any other sound abductive procedure which can deal with non-ground abducible atoms is -in principle- able to solve planning problems. Unfortunately, the theory of linear order causes intolerable efficiency problems. Not only the manipulation of this theory is computationally expensive, also SLDNFA will only generate linear *plans*, i.e. linearly ordered sets of events. In general, it would be desirable to get *partial plans* which satisfy the correctness criterion given in [Mis91a]: the goal state must be provable from each linearisation of the plan. Such a partial plan leaves open the order of independent events whose order is irrelevant for the final state. Compared to a partial planner, the number of plans computed by a linear planner is exponential in the number of independent pairs of events.

We propose a correct and more efficient solution for this problem. In the next section, we extend SLDNFA with a constraint solver for $\ll$. The resulting procedure is sound wrt the completion of the program together with the theory of linear order. The idea to use a constraint solver was borrowed from [MBD92] who implemented a constraint solver for the theory of *partial order*. Our module checks the satisfiability of the abduced $\ll$ atoms against the theory of linear order. The generated results are correct according to the criterion mentioned earlier: all linearisations of the partial order imply the query. The procedure is -to the best of our knowledge- the only current procedure that returns correct partial plans.

7.3 Extending SLDNFA for linear order

Below we assume the existence of an incomplete logic program P based on a language $\mathcal{L}$. The undefined predicates of $\mathcal{L}$ belong to one of the following disjunct classes: abducible predicates, strongly abducible predicates and linear order predicates. A linear order predicate is a binary predicate. We require that for each linear order $p/2$ there is a unary type predicate $\mathcal{U}_p/1$ in $\mathcal{L}$(two different linear orders may have the same type predicate). $\mathcal{U}_p/1$ may be defined or undefined. In Event Calculus, the linear order $\ll$ has type predicate $Time/1$. However, the procedure is not bound to temporal reasoning and can be used in any application

in which one or more linear orders occurs.

Definition 7.3.1 *We define the theory $\mathcal{L}inOrd$ as the set of FOL axioms with for each linear order predicate $p/2$:*

$$p(X,Y) \rightarrow \mathcal{U}_p(X) \wedge \mathcal{U}_p(Y)$$
$$p(X,Y), p(Y,Z) \rightarrow p(X,Z)$$
$$\leftarrow p(X,Y), p(Y,X)$$
$$\mathcal{U}_p(X) \wedge \mathcal{U}_p(Y) \rightarrow p(X,Y) \vee X = Y \vee p(Y,X)$$

The first axiom imposes a well-typedness condition on $p/2$. Then follow the axioms of transitivity and asymmetry (which subsumes irreflexivity), finally the axiom of linearity. This theory should not be added explicitly to the logic program when executing the program, but is implicit at the declarative level. Soundness and completeness results rely on $\mathcal{L}inOrd$.

Below we assume that the predicates of $\mathcal{L}$ are $p_1/n_1, \ldots, p_m/n_m$ where the first k predicates $p_1/2, \ldots, p_k/2$ are the linear orders. Given a set Δ of undefined ground atoms, Δ_{p_i/n_i} denotes the subset of Δ consisting of all p_i/n_i atoms.

Definition 7.3.2 *Given is a set of ground atoms $\Delta_{p/2}$ of a linear order $p/2$. $\mathcal{TC}(\Delta_{p/2})$ denotes the transitive closure of $\Delta_{p/2}$.*

Let Δ be a set of ground undefined atoms. $\mathcal{TC}(\Delta)$ denotes the following set:

$$\mathcal{TC}(\Delta_{p_1/2}) \cup \ldots \cup \mathcal{TC}(\Delta_{p_k/2}) \cup \Delta_{p_{k+1}/n_{k+1}} \cup \ldots \cup \Delta_{p_m/n_m}$$

Δ is LO-consistent iff $\mathcal{TC}(\Delta)$ contains no pair $p(t_1,t_2), p(t_2,t_1)$, for some linear order $p/2$.

We extend SLDNFA to SLDNFA-LO. When a ground atom $p(t_1,t_2)$ of a linear order is selected in a positive goal, the normal abduction operator of definition 5.3.3 is applied but in addition a new positive goal $\leftarrow \mathcal{U}_p(t_1), \mathcal{U}_p(t_2)$ is added. The situation when a linear order atom is selected in a negative goal is more complex. The extension is based on the following equivalence:

$$\begin{aligned}\forall(\leftarrow p(t_1,t_2), Q') \Leftrightarrow\ & \neg(\mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2)) \vee \\ & \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \wedge p(t_2,t_1) \vee \\ & \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \wedge t_1 = t_2 \vee \\ & \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \wedge p(t_1,t_2) \wedge \forall(\leftarrow Q')\end{aligned}$$

With each of the four disjuncts, one operator corresponds. The first operator assumes that t_1 or t_2 is not in $\mathcal{U}_p$ and adds a negative goal $\leftarrow \mathcal{U}_p(t_1), \mathcal{U}_p(t_2)$. The second assumes that $p(t_2,t_1)$ is true, the third that t_1 and t_2 are equal, the fourth that $p(t_1,t_2)$ is true but that the remainder of the goal $\leftarrow Q'$ fails.

Definition 7.3.3 *A selection is safe if it is not $(Q, \neg A)$ with A a non-ground atom and it is not (Q, A) with A a non-ground atom of a linear order.*

Definition 7.3.4 *The* positive linear order operator *(PLO-operator) applies when a linear order atom $p(t_1,t_2)$ is selected in a positive goal Q. Let Q' be obtained by deleting $p(t_1,t_2)$ in Q.*

The positive linear order operator produces the empty substitution, the abduced atom $p(t_1,t_2)$ and the positive goals Q' and $\leftarrow \mathcal{U}_p(t_1),\mathcal{U}_p(t_2)$. Formally:

$$\Delta' = \Delta \cup \{p(t_1,t_2)\}$$
$$\mathcal{PG}' = \mathcal{PG} \setminus \{Q\} \cup \{Q', \leftarrow \mathcal{U}_p(t_1),\mathcal{U}_p(t_2)\}$$
$$\mathcal{NG}' = \mathcal{NG}, \mathcal{NAG}' = \mathcal{NAG} \text{ and } \theta \text{ is } \varepsilon$$

The remaining operators apply when a linear order atom $p(t_1,t_2)$ is selected in a negative goal Q. Let Q' be the goal obtained by deleting $p(t_1,t_2)$ from Q.

The first negative linear order operator *(1st NLO-operator) produces the empty substitution and the negative goal $\leftarrow \mathcal{U}_p(t_1),\mathcal{U}_p(t_2)$. Formally:*

$$\mathcal{NG}' = \mathcal{NG} \setminus \{Q\} \cup \{\leftarrow \mathcal{U}_p(t_1),\mathcal{U}_p(t_2)\}$$
$$\mathcal{PG}' = \mathcal{PG}, \mathcal{NAG}' = \mathcal{NAG}, \ \Delta' = \Delta \text{ and } \theta \text{ is } \varepsilon$$

The second negative linear order operator *(2nd NLO-operator) produces the empty substitution, the abduced atom $p(t_2,t_1)$ and a positive goal $\leftarrow \mathcal{U}_p(t_1),\mathcal{U}_p(t_2)$. Formally:*

$$\mathcal{NG}' = \mathcal{NG} \setminus \{Q\}$$
$$\Delta' = \Delta \cup \{p(t_2,t_1)$$
$$\mathcal{PG}' = \mathcal{PG} \cup \{\leftarrow \mathcal{U}_p(t_1),\mathcal{U}_p(t_2)\}$$
$$\mathcal{NAG}' = \mathcal{NAG} \text{ and } \theta \text{ is } \varepsilon$$

The third negative linear order operator *(3rd NLO-operator) applies when $t_1 = t_2$ has positive unifier θ. In that case, it produces the substitution θ and the positive goal $\leftarrow \mathcal{U}_p(\theta(t_1))$. Formally:*

$$\mathcal{NG}' = \theta(\mathcal{NG} \setminus \{Q\})$$
$$\mathcal{PG}' = \theta(\mathcal{PG} \cup \{\leftarrow \mathcal{U}_p(t_1)\})$$
$$\mathcal{NAG}' = \theta(\mathcal{NAG}) \text{ and } \Delta' = \theta(\Delta)$$

The fourth negative linear order operator *(4th NLO-operator) produces the negative goal Q', the positive goal $\leftarrow \mathcal{U}_p(t_1),\mathcal{U}_p(t_2)$ and the abduced atom $p(t_1,t_2)$. Formally:*

$$\mathcal{NG}' = \mathcal{NG} \setminus \{Q\} \cup \{Q'\})$$
$$\Delta' = \Delta \cup \{p(t_1,t_2)\}$$
$$\mathcal{PG}' = \mathcal{PG} \cup \{\leftarrow \mathcal{U}_p(t_1),\mathcal{U}_p(t_2)\}$$
$$\mathcal{NAG}' = \mathcal{NAG} \text{ and } \theta \text{ is } \varepsilon.$$

Definition 7.3.5 *An SLDNFA-LO($^{\circ}_{+}$) derivation is defined as an SLDNFA($^{\circ}_{+}$) derivation, but using the additional set of operators. There is one additional condition: operators may be applied only on LO-consistent states. Hence, each Δ_i except the last is LO-consistent.*

An SLDNFA-LO($^{o}_{+}$) derivation is failed when an SLDNFA($^{o}_{+}$) derivation would be failed and in addition, when the last Δ_n is LO-inconsistent.

The definition of SLDNFA-LO($^{o}_{+}$) refutation remains identical to existing definitions, except that we require Δ_n to be LO-consistent.

It should be stressed that in many cases the above procedure can be seriously simplified. For example, assume programs and queries are range restricted, i.e. for each $p(t_1, t_2)$ atom in the body of a clause and in a query, there are atoms $\mathcal{U}_p(t_1), \mathcal{U}_p(t_2)$ in the body or in the query. In this case, the first NLO-operator needs not to be applied, since it will necessarily lead to failure. Also the addition of the positive goal $\leftarrow \mathcal{U}_p(t_1), \mathcal{U}_p(t_2)$ in all other operators is unnecessary. When moreover $\mathcal{U}_p$ is strongly abducible, then the operators can be further simplified. For two abduced atoms $\mathcal{U}_p(t_1), \mathcal{U}_p(t_2)$ with distinct t_1, t_2, there is always a negative goal $\leftarrow t_1 = t_2$. As a consequence when an atom $p(t_1, t_2)$ is selected and $t_1 \not\equiv t_2$ then there is a negative goal $\leftarrow t_1 = t_2$ and the third operator will necessarily lead to failure. When $t_1 \equiv t_2$, then the second and fourth operators lead to failure.

Another cause of inefficiency is the repeated test on the LO-consistency of Δ. Δ can become LO-inconsistent due to two different actions: the abduction of a fact $p(t_1, t_2)$ when $p(t_2, t_1)$ already belongs to $\mathcal{TC}(\Delta)$, and a positive unification yielding a substitution θ such that for some $p(t_1, t_2) \in \mathcal{TC}(\Delta)$, $\theta(t_1) = \theta(t_2)$. As a consequence, the test on LO-consistency needs only to be applied when a linear order atom is added to Δ or when skolem constants are unified. Moreover when $\mathcal{U}_p$ is strongly abducible, then for each atom $p(t_1, t_2) \in \Delta$, there is a negative goal $\leftarrow t_1 = t_2$. Hence, when t_1 and t_2 are unified, then failure will occur due to the negative goal. As a consequence, the LO-consistency test needs only to be applied when an atom $p(t_1, t_2)$ is abduced.

Here we consider such optimisations as an implementation issue and do not spend further attention to it. We illustrate the use of SLDNFA-LO with some examples.

Example A first example illustrates how SLDNFA-LO avoids bad derivations as in the light switch problem. To avoid lengthy derivations and goals, consider the simplified version:

$Time(e_1)$:-
$Time(e_2)$:-
light_off :- $e_1 \ll e_2$
light_off :- $e_2 \ll e_1$

Other procedures would succeed with the empty solution for the goal $\leftarrow \neg$ *light_off*. In essence, these procedures assume here that the light is on if neither $e_1 \ll e_2$ nor $e_2 \ll e_1$. SLDNFA-LO on the other hand fails on the goal because it realises that if not $e_1 \ll e_2$, then necessarily $e_2 \ll e_1$ (since by *FEQ*,

$e_1 \neq e_2$). The SLDNFA-LO tree is given in figure 7.1. In the figure, the "$\leftarrow$" operator in positive goals is replaced by "$+$", and in negative goals by "$-$".

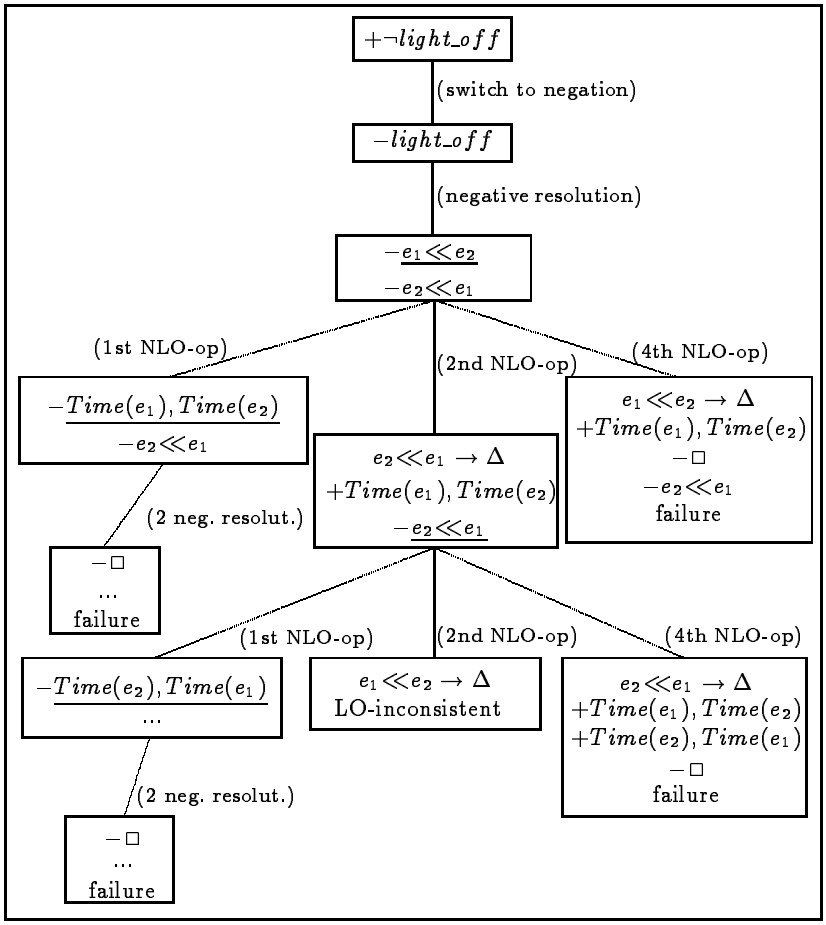

Figure 7.1: Failed SLDNFA-LO-tree for $\leftarrow \neg light\_off$

Example A second example illustrates how SLDNFA-LO can generate correct partial plans. Assume a planning problem with four fluents p, q, r, s and three types of actions. The first action initiates p, the second initiates q if s holds and the third initiates r and terminates s. Initially s is true, and in the goal state p, q and r are true. Clearly, a solution is given by abducing one event of each type with the constraint that the event of the second type should

precede the event of the third type. Without sacrificing the essence of the problem, we again simplify the program:

$Holds(P)$:- $Init(E, P)$
$Clip(P, E)$:- $Time(C), C \ll E, Term(C, P)$
$Init(e_0, p)$:-
$Init(e_1, q)$:- $\neg Clip(s, e_1)$
$Init(e_2, r)$:-
$Term(e_2, s)$:-
$Time(e_0)$:-
$Time(e_1)$:-
$Time(e_2)$:-

We present two derivations. They share the derivation in figure 7.2.

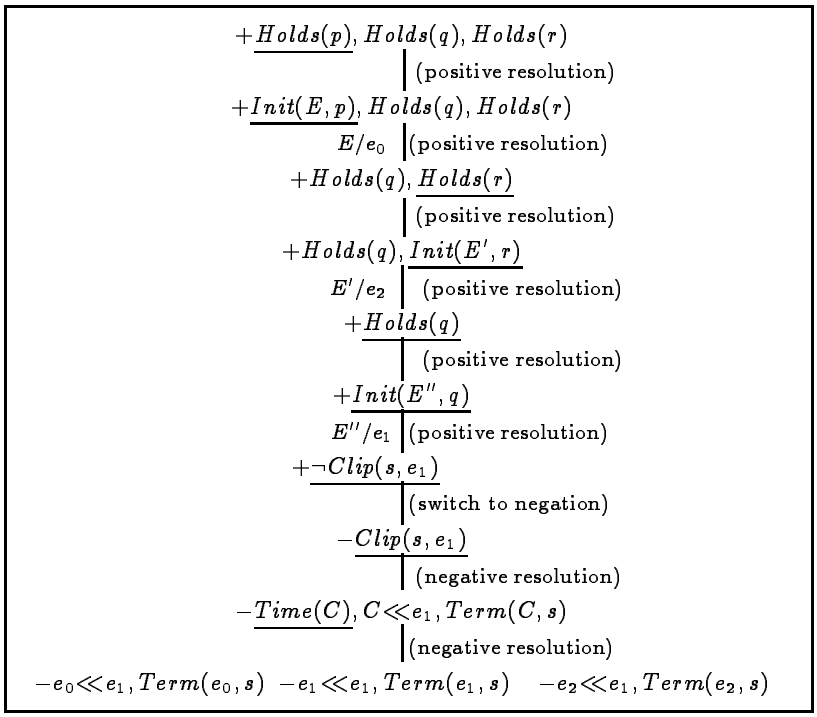

Figure 7.2: SLDNFA-LO-derivation for $\leftarrow holds(p), holds(q), holds(r)$

Note that in the negative goals $\leftarrow e_i \ll e_1, Term(e_i, s)$, only e_2 possibly terminates s. Depending on whether we select first the $Term(e_i, s)$ atom or $e_i \ll e_1$, we obtain more or less instantiated plans. The derivation in figure 7.3 selects $Term(e_i, s)$ atoms first and produces a partial plan $\Delta = \{e_1 \ll e_2\}$.

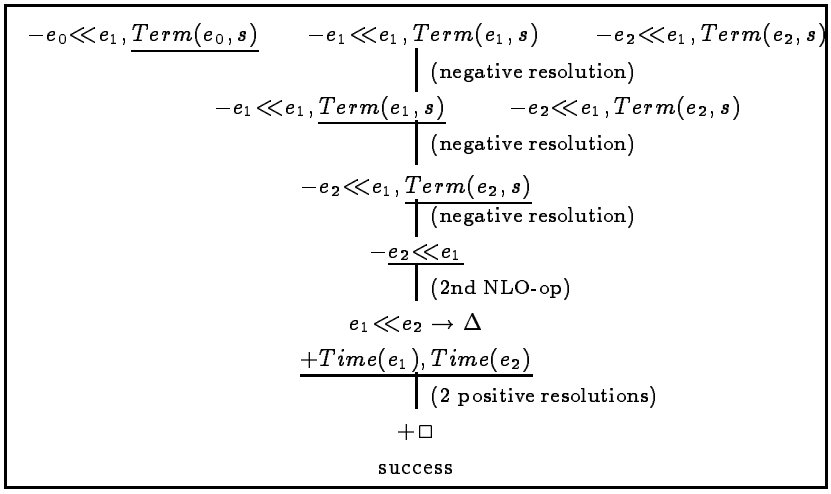

Figure 7.3: SLDNFA-LO-derivation for $\leftarrow holds(p), holds(q), holds(r)$

The derivation in figure 7.4 selects $e_i \ll e_1$ atoms first and produces a more instantiated plan $\Delta = \{e_1 \ll e_0, e_1 \ll e_2\}$. Another derivation exist which generates the alternative plan $\Delta = \{e_0 \ll e_1, e_1 \ll e_2\}$. This shows that different SLDNFA-LO refutations for the same query can lead to more or less instantiation of the plan. As a consequence, the degree of non-linearity of the plan depends on the computation rule.

Example Another example illustrates that SLDNF-LO is not always able to find solutions with least instantiated plans, not even with intelligent control. Consider the following example:

r :- p
r :- q
p :- $e_0 \ll e_1$
q :- $e_1 \ll e_0$
$Time(e_0)$:-
$Time(e_1)$:-

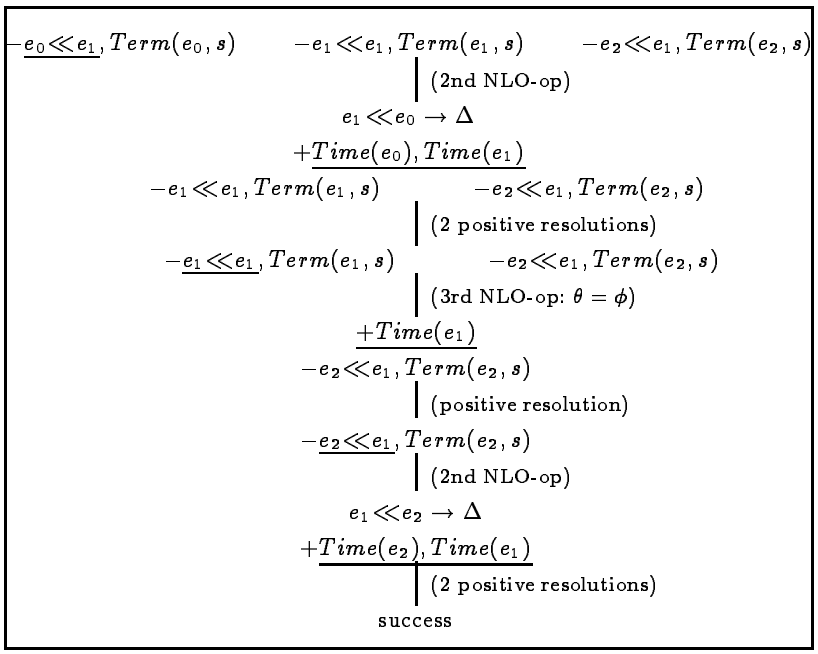

Figure 7.4: SLDNFA-LO-derivation for $\leftarrow holds(p), holds(q), holds(r)$

p holds when time e_0 precedes e_1. q holds when time e_1 precedes e_0. r holds when p or q holds. Clearly r holds whatever is the order of e_0 and e_1. Hence the empty solution Δ is a correct partial plan. However, SLDNFA-LO generates two solutions $\{e_0 \ll e_1\}$ and $\{e_1 \ll e_0\}$ under any computation rule. Figure 7.5 gives an example of a derivation.

The definitions of proof tree, computable children, SLDNFA-tree, state and explanation formula remain unaltered. The computable children for a given selection are the same as for pure SLDNFA, except when a negative goal and an atom of a linear order is selected. In that case, there are four computable children.

Below, we will split up a set Δ in the set Δ_{LO} of linear order atoms and the rest Δ_r. $\mathcal{LO}\text{-}comp(\Delta)$ is defined as $\mathcal{L}inOrd + comp_3(\mathcal{L} + Sk(\Delta), P + \Delta_r) + \Delta_{LO}$.

Theorem 7.3.1 *The SLDNFA-LO($^{\circ}_{+}$) procedure is sound in the following sense. Let (Δ, θ) be the result of an SLDNFA-LO($^{\circ}_{+}$) refutation for a goal Q_0. We have:*

$$<\mathcal{L} + Sk(\Delta), \mathcal{LO}\text{-}comp(\Delta)> \models \forall(\theta(\&(Q_0)))$$

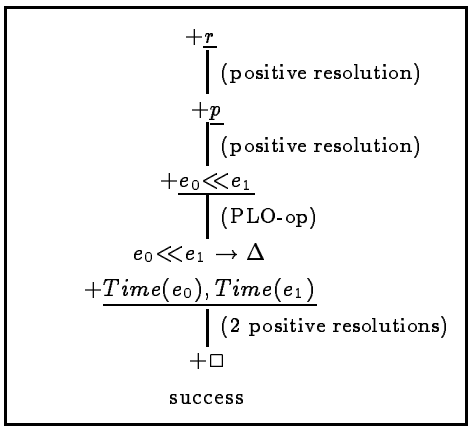

Figure 7.5: SLDNFA-LO-derivation for $\leftarrow holds(p), holds(q), holds(r)$

Moreover, Δ is well-typed: for any linear order atom $p(t_1, t_2) \in \Delta$:

$$\mathcal{LO}\text{-}comp(\Delta) \models \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2)$$

SLDNFA-$LO(^{\circ}_{+})$ is complete in the sense similar to theorems 5.6.1, 5.7.1 or 5.7.2 depending on the type of procedure that is applied. The only difference is that $\mathcal{LO}rd$ must be added to $comp_3(P)$.

Proof We must fit in the five new operators in the proof of soundness and completeness of SLDNFA. This involves proving successively the correctness of the operators, of the proof tree, of the soundness, of the explanation formula and finally of the completeness results.

Let Q be a positive goal in which an linear order atom $L_m = p(t_1, t_2)$ is selected and Q' is obtained by deleting L_m from Q. For the soundness and completeness, the following formula is important:

$$\mathcal{LO}rd \models \forall(\&(Q) \Leftrightarrow \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \wedge p(t_1, t_2) \wedge \&(Q'))$$

It follows trivially from the formula:

$$p(t_1, t_2) \Leftrightarrow p(t_1, t_2) \wedge \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2)$$

which is directly implied by $\mathcal{LO}rd$:

Let Q be a negative goal in which a linear order atom $L_m = p(t_1, t_2)$ is selected and Q' is obtained by deleting L_m from Q. For the completeness, the following formula is important:

$$\begin{aligned}\mathcal{L}in\mathcal{O}rd \models \forall(\overline{\nabla}(Q)) \Leftrightarrow\ & \neg(\mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2)) \vee \\ & \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \wedge p(t_2, t_1) \vee \\ & \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \wedge t_1 = t_2 \vee \\ & \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \wedge p(t_1, t_2) \wedge \forall(\overline{\nabla}(Q'))\end{aligned}$$

This formula follows from:

$$\begin{aligned}\neg p(t_1, t_2) \Leftrightarrow\ & \neg(\mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2)) \vee \\ & \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \wedge p(t_2, t_1) \vee \\ & \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \wedge t_1 = t_2\end{aligned}$$

which follows easily from $\mathcal{L}in\mathcal{O}rd$.

From the equivalence the following implications can be derived. They are important for the soundness of the four negative linear order operators:

$$\begin{aligned}&\mathcal{L}in\mathcal{O}rd \models \forall(\overline{\nabla}(Q)) \Leftarrow \neg(\mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2)) \\ &\mathcal{L}in\mathcal{O}rd \models \forall(\overline{\nabla}(Q)) \Leftarrow p(t_2, t_1) \wedge \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \\ &\mathcal{L}in\mathcal{O}rd \models \forall(\overline{\nabla}(Q)) \Leftarrow t_1 = t_2 \wedge \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2) \\ &\mathcal{L}in\mathcal{O}rd \models \forall(\overline{\nabla}(Q)) \Leftarrow p(t_1, t_2) \wedge \forall(\overline{\nabla}(Q'))\end{aligned}$$

From these implications the correctness of the proof tree under $\mathcal{LO}$-*comp*(Δ) can be proven without problems. There is one point worth noticing: the theory $\mathcal{LO}$-*comp*(Δ) does not contain the completed definition of the linear order predicates. One can easily check that theorem 5.5.1 relies on the completion of an abductive predicate only to prove that the $\mathcal{NAG}$ goals hold. Since $\mathcal{NAG}$ does not contain goals with a selected linear order atom, the completed definition of linear order predicates needs not to be added. The completed definition of Δ_{LO} should never be added: when Δ_{LO} is only a partial order, the completion of Δ_{LO} is inconsistent with $\mathcal{L}in\mathcal{O}rd$.

From the correctness of the proof tree, the soundness result:

$$<\mathcal{L} + Sk(\Delta), \mathcal{LO}\text{-}comp(\Delta)> \models \forall(\theta(\&(Q_0)))$$

follows directly. It is easily verified that Δ is well-typed: for each abduced atom $p(t_1, t_n)$, the atoms $\mathcal{U}_p(t_1)$ and $\mathcal{U}_p(t_2)$ occur in positive goals.

The correctness of the explanation formula wrt $\mathcal{L}in\mathcal{O}rd + comp_3(P)$ and the completeness results can be proven in a totally analogous way as in theorems 5.6.2, 5.6.1, 5.7.1 or 5.7.2. A small point to make is that when an SLDNFA-LO derivation K in the SLDNFA-LO tree fails due to an LO-inconsistency, then obviously the corresponding state formula of K is equivalent with false under $\mathcal{L}in\mathcal{O}rd$. □

When using SLDNFA-LO for planning, typically the predicate $Time/1$ and $\ll$ are undefined. In such a case, if SLDNFA-LO generates a solution Δ, then linearisations of $\Delta_{\ll}$ wrt to the set Δ_{Time} can be computed. Below we prove that such linearisations are correct solutions for the planning problem.

Definition 7.3.6 *Assume that the type predicate $\mathcal{U}_p$ of each linear order $p/2$ is undefined. Let Δ be a set of ground undefined atoms.*

A linearisation Δ' of Δ is a set:

$$\Delta'_{p_1/2} \cup \ldots \cup \Delta'_{p_k/2} \cup \Delta_{p_{k+1}/n_{k+1}} \cup \ldots \cup \Delta_{p_m/n_m}$$

where for $1 \leq i \leq k$: $\Delta_{p_i/2} \subseteq \Delta'_{p_i/2}$ and $\Delta'_{p_i/2}$ is a linear order on $\{t \mid \mathcal{U}_{p_i}(t) \in \Delta_{\mathcal{U}_{p_i}}\}$.

Proposition 7.3.1 *Assume that each type predicate of a linear order predicate is undefined in P. Assume that SLDNFA-LO generates a solution (Δ, θ) for a query Q. Let Δ' be any linearisation of Δ. It holds that:*

$$<\mathcal{L} + Sk(\Delta), comp_3(\mathcal{L} + Sk(\Delta), P + \Delta')> \models \forall(\theta(\&(Q_0)))$$

Proof Define $\mathcal{T}_1 = comp_3(\mathcal{L} + Sk(\Delta), P + \Delta')$ and $\mathcal{T}_2 = \mathcal{LO}\text{-}comp(\Delta)$. It suffices to prove that $\mathcal{T}_1$ entails $\mathcal{T}_2$. $\mathcal{T}_2$ is of the form:

$$\mathcal{L}in\mathcal{O}rd + comp_3(\mathcal{L} + Sk(\Delta), P + \Delta_r) + \Delta_{LO}$$

$\mathcal{T}_1$ includes $comp_3(\mathcal{L} + Sk(\Delta), P + \Delta_r)$ and $\mathcal{T}_1$ entails Δ_{LO}. Because Δ' defines linear orders on the types of the linear orders, $\mathcal{T}_1$ entails $\mathcal{L}in\mathcal{O}rd$.

□

A remaining question is whether for a generated solution Δ, $\mathcal{LO}\text{-}comp(\Delta)$ is consistent. This is not the case in general. Consider the following program P with linear order $p/2$ and associated type predicate $\mathcal{U}/1$:

$\mathcal{U}(t_1)$:- $\mathcal{U}(X), \neg p(X, t_1), \neg p(t_1, X), \neg X = t_1$
$\mathcal{U}(t_2)$:-

The query $\leftarrow \mathcal{U}(t_2)$ succeeds with empty Δ. $\mathcal{LO}\text{-}comp(\phi)$ is not consistent. Indeed, it is easy to see that $\mathcal{U}(t_1)$ is true iff there exists some element x of $\mathcal{U}$, different from t_1 such that x and t_1 are not related by $p/2$. So, when $\mathcal{U}(t_1)$ is true, then the axiom of linearity is not satisfied. Assume that $\mathcal{U}(t_1)$ is not true. In that case, for all elements x of $\mathcal{U}$, $p(x, t_1)$ and $p(t_1, x)$ are false (otherwise the typing axiom is violated). Then $\mathcal{U}(t_1)$ has the strong direct direct justification $\{\mathcal{U}(t_2), \neg p(t_2, t_1), \neg p(t_1, t_2), \neg t_2 = t_1\}$. This is a contradiction.

In practice, such ill-defined programs do not occur. Below we give a large class of programs which do not suffer from the inconsistency problem. The theorem is based on the following lemma.

Lemma 7.3.1 *Let M be a 3-valued Herbrand interpretation, which is 2-valued on the linear orders. Assume that M is LO-consistent. Assume that for each linear order atom $p(t_1, t_2) \in M$, it holds that $M \models \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_1)$. There exists an extension M' of M with identical interpretation of the non-linear order predicates, which is at least a weak model of $\mathcal{L}in\mathcal{O}rd$.*

Proof The proof is based on the mathematical property that any partial order on a set of objects can be extended to a linear order on this set. Since M is LO-consistent and well-typed, M defines partial orders on the sets $\{t \mid \mathcal{H}_M(\mathcal{U}_{p_i}(t)) \geq \mathbf{u}\}$. Hence, it is possible to extend M to M' by extending all partial orders to linear orders on these sets. Obviously the resulting extension is at least a weak model of $\mathcal{L}in\mathcal{O}rd$. It may be weak due to instances of the well-typedness axiom. When $\mathcal{U}_p(t_1)$ is unknown and $p(t_1, t_2)$ is true, then $p(t_1, t_2) \rightarrow \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_1)$ is unknown. □

Theorem 7.3.2 *Given is an incomplete program such that for each linear order $p/2$ and associated type predicate $\mathcal{U}_p/1$ one of the following conditions is satisfied:*

- *$\mathcal{U}_p$ is undefined, or*
- *the definition of $\mathcal{U}_p$ does not depend on the linear order predicates.*

Then for any generated solution Δ, $<\mathcal{L} + Sk(\Delta), \mathcal{LO}\text{-}comp(\Delta)>$ is consistent.

Proof Let (Δ, θ) be a solution generated by SLDNFA-LO. Split up Δ in Δ_{LO} and Δ_r as before. In 5 successive stages, we gradually extend incomplete Herbrand interpretations to a model of $\mathcal{LO}\text{-}comp(\Delta)$. These extension are often based on theorem 4.3.4 which guarantees that any incomplete interpretation for undefined predicates can be extended to a directly justified model of a set of definitions or equivalently to a model of the completions of these definitions.

- In the first stage, we select an incomplete Herbrand interpretation M_1 for the abducible and strongly abducible predicates, such that $comp_3(\Delta_r)$ is satisfied. By theorem 4.3.4, such an incomplete interpretation always exists.
- In the second stage, we extend M_1 to an incomplete Herbrand interpretation M_2 for all defined predicates in P which do not depend on linear order predicates. M_2 is selected such that it satisfies $comp_3(P')$. Here P' is the subset of P with all definitions of predicates which do not depend on linear order predicates. By theorem 4.3.4, such an extension of M_1 can always be found.
- In the third stage, M_2 is extended to M_3 for each linear order $p/2$ such that $M_3(p/2) = \Delta_{p/2}$. Obviously this is always possible.

- In the fourth stage, the interpretation of the linear orders in M_3 is extended to obtain an incomplete interpretation M_4 which satisfies $\mathcal{L}in\mathcal{O}rd$. We show that M_3 satisfies the conditions of lemma 7.3.1. $\Delta_{p/n}$ is LO-consistent. For each $p(t_1, t_2) \in \Delta$, it holds that $\mathcal{LO}\text{-}comp(\Delta) \models \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2)$ (theorem 7.3.1). When $\mathcal{U}_p$ is undefined, then clearly $\mathcal{U}_p(t_1), \mathcal{U}_p(t_2) \in \Delta_r$, and hence $M_3 \models \mathcal{U}_p(t_1), \mathcal{U}_p(t_2)$. When $\mathcal{U}_p$ is defined, then since this predicate does not depend on the linear orders and since M_3 satisfies the completed definitions of $\mathcal{U}_p$ and all the predicates on which U_p depends, $M_3 \models \mathcal{U}_p(t_1) \wedge \mathcal{U}_p(t_2)$. From the lemma it follows that M_3 can be extended to at least a weak model of $\mathcal{L}in\mathcal{O}rd + \Delta_{LO}$.
- In the final stage, M_4 is extended to M for the remaining predicates: the defined predicates which depend on linear orders. M is selected such that it satisfies $comp_3(P'')$, where $P'' = P \setminus P'$.

We obtain an interpretation of $\mathcal{L} + Sk(\Delta)$ which satisfies the following theories:

$comp_3(\Delta_r)$, $comp_3(P')$, Δ_{LO}, $\mathcal{L}in\mathcal{O}rd$, $comp_3(P'')$

The union of these theories is precisely $\mathcal{LO}\text{-}comp(\Delta)$. □

7.4 Representing temporal domains

For many applications, the version of event calculus introduced in section 7.2 is too simplistic. A number of extensions has been proposed in the literature. Below we discuss some of these extensions, and new ones.

7.4.1 Incomplete knowledge: other examples

Another example with incomplete knowledge is the stolen car problem [Bak89]: *initially, I leave my car in the garage; two days later, the car is gone, stolen; cars are only stolen during the night; when was it stolen?*. Since initial situation and events are known, it suffices to have $\ll$ as undefined predicate. The domain is represented by the following rules:

$Terminates(E, i\_have\_car) \leftarrow Act(E, steal)$
$Initiates(E, night) \leftarrow Act(E, enter\_night), \neg Holds\_at(night, E)$
$Terminates(E, night) \leftarrow Act(E, enter\_day), Holds\_at(night, E)$
$Initially(i\_have\_car) \leftarrow$

In addition, there are integrity constraints:
$Time(start), Time(e_1), Time(e_2), Time(e_3), Time(e_4), Time(t_{end})$,
$Act(e_1, enter\_night), Act(e_2, enter\_day), Act(e_3, enter\_night)$,
$Act(e_4, enter\_day)$,

$start \ll e_1, e_1 \ll e_2, e_2 \ll e_3, e_3 \ll e_4, e_4 \ll t_{end}$
$Act(E, enter\_night) \rightarrow E = e_1 \vee E = e_3$
$Act(E, enter\_day) \rightarrow E = e_2 \vee E = e_4$
$Act(E, steal) \rightarrow Holds(night, E)$

This event calculus has models in which a *steal* event occurs during the first night, and another in which the *steal* event occurs during the second night. Note that it is necessary to close the *enter_night* and *enter_day* events by the constraints:

$Act(E, enter\_night) \rightarrow E = e_1 \vee E = e_3$
$Act(E, enter\_day) \rightarrow E = e_2 \vee E = e_4$

In this case, we know that there are only two nights, so this partial knowledge must be expressed by integrity constraints about $Act/2$. Without these constraints, there would be models in which more than two nights occurred and in some of these models, the car would be stolen during other nights.

SLDNFA-LO solves the query $\leftarrow \neg Holds\_at(i\_have\_car, t_{end}), \neg false$ by abducing a *steal* event which occurs either between e_1 and e_2 or between e_3 and e_4. Without the constraints on *enter_night* and *enter_day*, SLDNFA generates erroneous solutions in which the car is stolen during some third night.

Temporal reasoning problems are often classified according to the direction in time in which reasoning is necessary. In a *prediction problem*, the goal is to derive the final state or to prove some property about the final state. To prove that the turkey is dead at t_{end} in the Yale Turkey Shooting problem is a prototypical prediction problem. An ambiguous prediction problem is one in which there is more than one possible final state. In subsection 7.4.4, an example is given. In a *postdiction problem*, the goal is to derive properties about the initial state and/or about earlier events, given information about some final state. An example is the Murder Mystery. An ambiguous postdiction problem occurs when different initial states or sets of events lead to the observed final state. The stolen car problem is an example. Observe that in incomplete event calculus, a general purpose reasoning procedure like SLDNFA-LO can be used both for (ambiguous) prediction and postdiction.

The terminology *(ambiguous) prediction* and *postdiction* does not really characterise a temporal domain in a declarative way. Rather it characterises a type of temporal problem to be solved. For example, on an event calculus representing a planning domain, there may be both ambiguous postdiction problems and ambiguous prediction problems to be solved. The search for a plan which produces some final state is an ambiguous postdiction problem. To prove that in some planning domain, some state cannot occur is an ambiguous prediction problem. As a consequence, the terminology is not suited to classify temporal domains on a declarative basis.

7.4.2 Events versus Actions

Whereas the event calculus introduced in section 7.2 is based on time, in [KS86] and in all other versions, event calculus is based on the notion of *event*. An event can be seen as a specific occurrence of some action. Vice versa, actions can be viewed as *event types*. In the YTS example in section 7.2, events have no explicit representation: *loading*, *waiting* and *shooting* denote event types, not events. Events are implicitly represented in the three facts $Act(e_1, loading)$, $Act(e_2, waiting)$ and $Act(e_3, shooting)$.

Events can easily be introduced in our variant of event calculus. For example, the YTS problem can easily be re-implemented on the basis of events. The language is extended with constants ev_1, ev_2 and ev_3, and a predicate $Event\_Type/2$. The incomplete logic program is modified by deleting the definitions of $Initiates/2$, $Terminates/2$ and $Act/2$ and adding the following definitions:

$Initiates(T, loaded)$:- $Act(T, E), Event\_Type(E, loading)$
$Terminates(T, alive)$:- $Act(T, E), Event\_Type(E, shooting),$
$\qquad\qquad\qquad\qquad Holds\_at(loaded, T)$
$Terminates(T, loaded)$:- $Act(T, E), Event\_Type(E, shooting)$
$Act(e_1, ev_1)$:-
$Act(e_2, ev_2)$:-
$Act(e_3, ev_3)$:-
$Event\_Type(ev_1, loading)$:-
$Event\_Type(ev_2, waiting)$:-
$Event\_Type(ev_3, shooting)$:-

There are several arguments why the explicit representation of events is in general preferable. One argument, expressed by [KS86], [Kow92], [KS92] is that in general, complex objects such as events, which are involved in a large number of relations, are best represented explicitly, and their relations are best represented by binary predicates. For example, a promotion event of a person in a firm has attributes: the person who promotes, the new rank, the old rank, the time of the promotion, the person who orders the promotion, the motivation for the promotion (*hard_working*, *capable*, *longtime_employed*), etc.. One way to represent such a complex action is by introducing a functor with arguments for each of the attributes. However, such a representation is less adequate when there are unknown attributes or when some attributes are not functional. Moreover, a fixed arity representation cannot cope gracefully with the range of descriptions that can be expected for an event. A solution to this problem is by making explicit the notion of event and describing it by binary predicates, or a syntactic variant of binary relations: by using attributes in an object oriented style as in [KS92].

A second argument is that the explicit representation of events allows to express situations in which more than one event of the same type occur at the same moment. Two atomic rules $Act(e_3, shooting)$:- still represent only one shooting

event. On the other hand, in an event based representation, we can represent this by:

$Act(e_3, ev_{31})$:-
$Act(e_3, ev_{32})$:-
$Event\_Type(ev_{31}, shooting)$
$Event\_Type(ev_{32}, shooting)$

The extra expressivity may be important in situations when the effect of two distinct events is not equivalent with the effect of the two isolated events. For example, substitute an elephant for the turkey in the YTS problem. Assume that there must be at least two shooting events at the same time to kill the elephant. This can be expressed by:

$$Terminates(T,alive) \text{ :- } Act(T, Ev_1), Act(T, Ev_2), \neg Ev_1 = Ev_2, \\ Event\_Type(Ev_1, shooting), Event\_Type(Ev_2, shooting)$$

It should be stressed however that both arguments are not specific for the notion of event but hold in general for all complex concepts. Or, the issue of representing events versus actions stands orthogonal on the representation of temporal domains.

When events are explicitly represented, it can be interesting to use a modified representation of the frame axiom which was presented in [Sha90]:

$$Holds\_at(P, T) \text{ :- } Happens(E), Time\_of(T_E, E), T_E \ll T, \\ Initiates(E, P), \neg Clipped(T_E, P, T)$$
$$Clipped(T_1, P, T_2) \text{ :- } Happens(C), Time\_of(T_C, C), In(T_C, T_1, T_2), \\ Terminates(C, P)$$

$Happens/1$ is a type predicate representing events. $Time\_of/2$ is nothing than a renaming of $Act/2$. The intended interpretation of $Initiates/2$ and $Terminates/2$ is slightly different from the original version: the first argument is an event with an initiating or terminating effect. An event is bound to a specific time and a specific action. Therefore, we have the following integrity constraints:

$Happens(E) \rightarrow \exists T : Time\_of(T, E)$
$Happens(E), Time\_of(T_1, E), Time\_of(T_2, E) \rightarrow T_1 = T_2$
$Event\_Type(E, A1), Event\_Type(E, A2) \rightarrow A1 = A2$

These constraints are either entailed by the program or can be explicitly added to the program.

A further optimisation is to drop time completely, and to express $\ll$ directly on the events. Three possible time relations are possible between two distinct events e_1, e_2: either $e_1 \ll e_2$, e_1 and e_2 concur or $e_2 \ll e_1$. By forbidding simultaneous events, the resulting variant of event calculus is the one proposed in [Sha89], [Mis91a], [DMB92]. This version is syntactically equivalent with the version in section 7.2: it is obtained by substituting $Happens/1$ for $Time/1$.

7.4.3 Pre-conditions and context dependent effects of actions

In general, the effect of an event will depend on the state or the context in which it occurs. A strong form of dependence occurs when the event relies on some *necessary preconditions*. E.g. a robot can only pick up some object if the robot is free, if the object is not fixed to the ground, etc.. [Sha89] and [Mis91a] proposed a general way to formulate necessary pre-conditions. They introduced a predicate $Succeeds/1$ which takes events as argument. The intended interpretation is that the event succeeds, i.e. that its necessary preconditions are satisfied. The law of inertia is modified as follows:

$$Holds\_at(P,T) \text{ :- } Happens(E), \underline{Succeeds(E)}, Time\_of(T_E,E), T_E \ll T, \\ Initiates(E,P), \overline{\neg Clipped}(T_E,P,T)$$
$$Clipped(T_1,P,T_2) \text{ :- } Happens(C), \underline{Succeeds(E)}, Time\_of(T_C,C), \\ In(T_C,T_1,T_2), \overline{Terminates}(C,P)$$

$Succeeds/1$ is defined by domain dependent rules. For example, we wish to express that a loading event succeeds always and that a shooting event succeeds only when the gun is loaded. The solution is:

$$Succeeds(E) \text{ :- } Event\_Type(E,shooting), Holds\_at(loaded,E)$$
$$Succeeds(E) \text{ :- } Event\_Type(E,loading)$$

Instead of adding $Succeeds/1$ atoms to the inertia axiom, an alternative solution which was not proposed so far, is to add an integrity constraint:

$$Happens(E) \rightarrow Succeeds(E)$$

A weaker form of dependence occurs when the effect of an event depends on what is true or not true at the moment of the event. Such an effect is called a *context dependent effect*. A simple example of a context dependent effect is given in the YTS example: the terminating effect of *shooting* on *alive* depends on *loaded*. Another example of a context dependent effect is in a block world: when the robot picks some block, then the underlying block becomes $free$:

$$Initiates(T,free(B)) \text{ :- } Act(T,pick(B1)), Holds\_at(on(B1,B),T)$$

7.4.4 Indeterminate events

Incomplete event calculus allows to represent *indeterminate actions*. A classical example appears in the Russian Turkey Shooting problem [San91]: *initially a turkey is alive, a gun is unloaded; there is a loading event, followed by an event of spinning the gun's chamber, and finally a shooting event.* The effect of the spinning event is indeterminate: the event possibly unloads the gun. As in section 6.7, the problem can be solved by introducing an undefined $good\_luck/1$ predicate. The effect of spinning can be expressed as follows:

$$Terminates(T, loaded) \text{ :- } Act(T, spinning), good\_luck(T)$$

A complete theory for the Russian Turkey Shooting problem is obtained from the YTS program by adding this clause and substituting the atomic clause

$$Act(e_2, spinning) \text{ :-}$$

for the clause

$$Act(e_2, waiting) \text{ :-}$$

The resulting program has different models. In one model, $good\_luck(e_2)$ is true and the turkey is alive at t_{end}. In another model, $good\_luck(e_2)$ is false and the turkey is dead at t_{end}. The RTS problem is considered as a typical *ambiguous prediction* problem: the final state is not uniquely determined by the problem description.

Note that the program has only $good\_luck/1$ as undefined predicate. No undefined linear order predicates appear in it and hence, SLDNFA can be applied here. SLDNFA solves the goal $\leftarrow Holds\_at(alive, t_{end})$ by returning $\Delta = \{good\_luck(e_2)\}$. It solves $\leftarrow \neg Holds\_at(alive, t_{end})$ by returning the empty solution, representing the situation in which the turkey has no *good_luck* at e_2. Together with this Δ, a negative abductive constraint $\leftarrow good\_luck(e_2)$ is computed. SLDNFA can also be used for deduction. For example, the theory entails $\neg good\_luck(e_2) \leftarrow \neg Holds\_at(alive, t_{end})$. The negation of this implication is given by $good\_luck(e_2) \wedge \neg Holds\_at(alive, t_{end})$. SLDNFA proves the inconsistency of this formula by failing on the query: $\leftarrow good\_luck(e_2), \neg Holds\_at(alive, t_{end})$.

7.4.5 The ramification problem

An important part of the frame problem is the ramification problem [San91]. It is the problem of representing the effect of actions on logically related properties. When events affect some set of properties, this may have implicit *ramifications* or effects on the logically related properties.

The ramification problem in event calculus has been investigated in some depth by [Mis91a] and [Kow92]. Our investigation below extends their study in several aspects. A study of the ramification problem is interesting since it reveals a number of fundamental concepts in temporal reasoning.

[Mis91a] and [Kow92] argue that in the context of logic programming, the ramification problem often pops up in the presence of *derived properties*: properties which have a definition in terms of other properties. They showed that, contrary to situation calculus, event calculus can deal very easily with derived properties. We illustrate this by a simple extension of the YTS problem, in which the derived property *dead* is defined as follows:

$$Holds\_at(dead, T) \text{ :- } \neg Holds\_at(alive, T)$$

This clause is added to the YTS program in section 7.2. In all models of the resulting theory (wrt completion semantics), *dead* is true at t_{end}. Prolog succeeds on the goal $\leftarrow Holds\_at(dead, t_{end})$. Things become interesting when we add a new action *ressurection*, whose effect is described as follows:

$$Initiates(E, alive) \text{ :- } Act(E, ressurection)$$

If we add atomic clauses for each of the following atoms $Time(e_4)$, $e_3 \ll e_4$, (+transitive closure), $e_4 \ll t_{end}$ and $Act(e_4, ressurection)$, then the goal

$$\leftarrow Holds\_at(alive, t_{end})$$

succeeds and the goal

$$\leftarrow Holds\_at(dead, t_{end})$$

finitely fails. $Holds\_at(dead, E)$ is only provably true for $E = e_4$, just after the shooting. For all other time points it is provably false.

It is of interest to compare this solution with situation calculus. It is well-known that situation calculus fails to give the right answer in situations analogous with the resurrection example. The situation calculus solution for the YTS problem was given in section 6.2. It should be extended with the following normal clauses:

$$Holds(alive, Result(ressurection, S)) \text{ :-}$$
$$Noninertial(alive, ressurection, S) \text{ :-}$$
$$Holds(dead, S) \text{ :- } \neg Holds(alive, S)$$

The resulting program entails the following formulas:

$$Holds(dead, Result[loading; waiting; shooting, S_0])$$
$$Holds(dead, Result[loading; waiting; shooting; resurrection, S_0])$$

The first implication is intuitively correct and easy to prove. The second $Holds/2$ atom is erroneous. It can be derived via the following instance of the inertia law of situation calculus:

$$Holds(dead, Result[loading; waiting; shooting; ressurection, S_0]) \text{ :-}$$
$$Holds(dead, Result[loading; waiting; shooting, S_0]),$$
$$\neg Noninertial(dead, ressurection, Result[loading; waiting; shooting, S_0])$$

Both literals of the body are true. Note that $Noninertial(dead, X, Y)$ is provably false for each X and Y.

Compare the above instance with the instance of the inertia axiom of event calculus:

$$Holds(dead, t_{end}) \text{ :- } Time(E), E \ll t_{end}, Initiates(E, dead),$$
$$\neg Clipped(E, dead, t_{end})$$

The body cannot be true because $Initiates(E, dead)$ is provably false for all E. [Kow92] shows that the problem with situation calculus can easily be solved when it is possible to distinguish between *primitive fluents* and *derived fluents*. In that case, a new predicate $primitive/1$ can be introduced which is true for all primitive fluents. The law of inertia is modified as follows:

$$Holds(F, Result(A, S)) \text{ :- } Holds(F, S), primitive(S), \neg Noninertial(F, A, S)$$

The requirement of being able to distinguish between primitive and derived fluents is also imposed in the context of event calculus by [Mis91a]. As we will show below, this requirement can be relaxed.

Observe that the presence of derived properties forces us to revise the intended meaning of the predicates $Initiates/2$ and $Terminates/2$. In section 7.2, it was argued that for example $Initiates(E, P)$ holds iff the event E has an initiating effect on P. This is not true when derived properties are added. For example, the shooting event e_3 initiates *dead*, but $Initiates(e_3, dead)$ is provably false. Here, we should distinguish between *primitive* initiating and *derived* initiating (or terminating) effects. $Initiates/2$ represents only the primitive initiating effects of events (such as $Initiates(e_1, loaded)$). It does not represent the derived initiating effects such as the effect of e_3 on *dead*.

The ramification problem shows up not only with derived properties but also in more subtle form. An illustration is known in the literature as the Walking Turkey Shooting problem [Bak91]. The problem is a small extension of the YTS problem in which initially the turkey is not only alive but also walking. The problem is to adapt the YTS representation such that it can be derived that at t_{end} the turkey is dead and not walking.

Clearly we should try to represent the general law that a walking turkey is necessarily alive. The problem now is that there are different ways to do this. In a first attempt, we may choose, in analogy with *dead* in the example above, for the rule:

$$Holds\_at(alive, E) \text{ :- } Holds\_at(walking, E)$$

Unfortunately, this is not a correct solution. Prolog answers yes on both queries below:

$$\leftarrow Holds\_at(walking, t_{end})$$
$$\leftarrow Holds\_at(alive, t_{end})$$

The problem is caused by the fact that initially the turkey is walking and no event ever terminates this fluent. Hence, at t_{end} the turkey is still walking and from this, the new rule allows to derive that the turkey is still alive.

A second attempt is to weaken " :- " to "$\leftarrow$" in the above rule and to add the integrity constraint:

$$Holds\_at(alive, E) \leftarrow Holds\_at(walking, E)$$

Also this solution fails. The resulting program is inconsistent, since it implies that the turkey is not alive at t_{end}, but since nothing terminates *walking*, *walking* is still true at t_{end}. After transforming the integrity constraint to:

$$false \text{ :- } Holds\_at(walking, E), \neg Holds\_at(alive, E)$$

Prolog can prove $false$.

The problem with both solutions is that they fail in terminating *walking* when *shooting* occurs. The third solution offers a direct solution: since *alive* is a *necessary precondition* for being able to walk, any event which terminates *alive* terminates *walking*. This can be represented by:

$$Terminates(walking, E) \leftarrow Terminates(alive, E)$$

Also, any event which initiates *walking* has precondition *alive*. This is represented by adding $Holds\_at(alive, E)$ to the body of clauses for $Succeeds(E)$ or $Initiates(E, walking)$.

The resulting program allows to derive the correct answers. Note that the constraint formula $Holds\_at(alive, E) \leftarrow Holds\_at(walking, E)$ is implied by each of the three proposals. Despite that, only the third proposal is a solution.

Although the YTS problem with derived fluent *dead* and the WTS problem definitely show some potential of event calculus to solve the ramification problem, the solutions advanced here -especially the one for the WTS problem- are not based on general principles yet. Clearly, an ad hoc solution in which a user is forced to make a blind search for a correct representation of a given constraint (as we seemed to do for the WTS problem) is not very satisfactory. The success of event calculus as a general solution for the ramification problem and for temporal reasoning, will depend on whether a general methodology for representing different types of relations between fluents and events can be developed. As we show below, such a methodology might bring to the surface a number of fundamental issues in temporal reasoning.

Let us analyse the three above proposals for the WTS problem. The first proposal was based on the following analogy with *dead* in the YTS example:

not *alive* implies *dead*
walking implies *alive*

Looking closer however, it turns out that this analogy is only superficial. There is a subtle but important distinction between how not *alive* implies *dead* and how *walking* implies *alive*. This becomes obvious when judging the correctness of the following natural language sentences:

to be *dead* means to be not *alive*
to be *alive* means to be *walking*

The first sentence is correct: *dead* can be seen as a new name to denote the state of not being *alive*. The second sentence is clearly not correct.

There is also a difference between how *walking* implies *alive* and how terminating *alive* implies terminating *walking*. Evaluate the following sentences:

to terminate *alive* causes to terminate *walking*
to be *walking* causes to be *alive*

While the first sentence is correct, the second sentence is definitely not true.

What this analysis shows is that we should learn to differentiate between different forms of implication. Some "implications" are *definitions of new concepts* (e.g. *dead*). Other "implications" represent *causal laws* (e.g. the WTS). There exists also *pure implications*. An example is the formula:

$$Holds(alive, S) \leftarrow Holds(walking, S)$$

Clearly, this rule neither represents a causal law nor a definition for the concept of *alive* but nevertheless makes a correct statement.

Different types of implications should be represented in different ways. Pure implications such as the one in the previous paragraph should not be added to the program as an " :- " rule, but may be added as a FOL implication. The use of " :- " to represent (constructive) definitions was advocated before in chapter 4, and returns here for *dead*. The interpretation of " :- " to represent *causal laws* is new. Intuitively, it seems that a causal law relates causes to an effect in a similar way how a clause in a constructive definition relates more primitive concepts in the body to the defined concept in the head. An essential property of causality is that phenomena do not cause themselves. This corresponds to requirement for a constructive definition that a concept should not be defined in terms of itself. This issue seems worth to be explored further in the future. It would explain why logic programs such as event or situation calculus have sometimes such a natural reading (compared with, for example, the completed definitions or the type of formulas used in [Rei92]).

Below we illustrate the above concepts in a more complex example: *in a game playing family, somebody is sad if one of her ancestors is sad or otherwise, if she has just lost a game; a person is not sad if she just won a game and each of her ancestors is not sad.* In this example, there is clearly a causal relationship between the sadness of an ancestor and of her descendants. With respect to the initiating and terminating effects of winning and loosing a game, we can make a distinction between primitive and derived effects. Loosing a game has a primitive initiating effect on sad for the person who is loosing and a derived initiating effect on sad for her descendants. Winning a game has a primitive terminating effect on sad for the person who is winning and a derived terminating effect for her descendants.

The logic of this problem is represented as follows:

$Holds\_at(sad(X), T)$:- $Ancestor(Y, X), Holds\_at(sad(Y), T)$
$Initiates(E, sad(X))$:- $Act(E, loose\_game(X))$
$Terminates(E, sad(X))$:- $Act(E, win\_game(X))$

Each of these sentences represent causal laws. Note that in this example, no distinction between primitive and derived fluents can be drawn. A problem specification is given by adding an atomic clause for each atom of the following list:

$Time(e_1), Act(e_1, loose\_game(daisy))$,
$Time(e_2), Act(e_2, loose\_game(george)), e_1 \ll e_2$
$Time(e_3), Act(e_3, win\_game(sue)), e_1 \ll e_3, e_2 \ll e_3$,
$Time(e_4), Act(e_4, win\_game(daisy)), e_1 \ll e_4, e_2 \ll e_4, e_3 \ll e_4$
$Time(t_{end}), e_1 \ll t_{end}, e_2 \ll t_{end}, e_3 \ll t_{end}, e_4 \ll t_{end}$

and atomic clauses for each ancestor relation represented in the graph in figure 7.6.

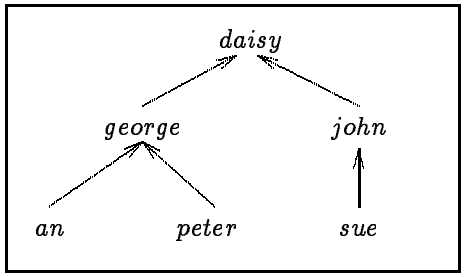

Figure 7.6: Ancestor graph

Prolog succeeds on all following queries:

$\leftarrow Holds\_at(sad(sue), e_3)$
$\leftarrow Holds\_at(sad(sue), e_4)$
$\leftarrow \neg Holds\_at(sad(sue), t_{end})$
$\leftarrow Holds\_at(sad(peter), t_{end})$

Notice that the event e_3 with primitive terminating effect on $sad(sue)$ has no effect on *sue*'s sadness. This is due to the fact that $Holds\_at(sad(sue), e_4)$ is derivable via the rule for sadness and the fact that *daisy* is still *sad*. This can be explained as follows: the true cause for *sue*'s sadness is that *daisy* is *sad*. The primitive terminating effect does not remove this cause, and hence *sue* remains *sad*. This is the wanted behaviour.

A variant of the problem is when a family member is always glad after winning a game but becomes sad when she looses a game or when an ancestor becomes sad.

Now, the effect of an ancestor loosing a game is a primitive effect on the temper of his descendants. The logic of this problem is represented as follows:

$Initiates(E, sad(X))$:- $Act(E, loose\_game(X))$
$Initiates(E, sad(X))$:- $Ancestor(Y, X), Initiates(E, sad(Y))$
$Terminates(E, sad(X))$:- $Act(E, win\_game(X))$

In this formulation of the problem, a person becomes glad when she wins a game, and remains glad until she or one of her ancestors looses a game. Taking the same problem specification as above, now the goal:

$\leftarrow Holds\_at(sad(sue), e_4)$

fails. Despite the fact that *daisy* is still sad, *sue* is glad at e_4.

The subtle differences between the two variants become more apparent when judging the correctness of the following sentences in both specifications:

an ancestor to be sad causes a descendant to be sad
an ancestor to become sad causes a descendant to become sad

In the first variant of the game playing family, the first sentence is represented, while in the second variant, the second is correct. The example illustrates that to represent some temporal domain, one needs to identify the causal relationships and definitions. These relationships must be represented by " :- " clauses.

It seems justified to conclude that event calculus has potential to represent derived properties and ramification. However, the representational problems are subtle. In the future, more experiments of the kind presented in this section are necessary and should be generalised to come to a general methodology for dealing with logically related properties in complex worlds. In this section we have indicated some possibly important concepts in such a methodology: different types of implications such as definitions, causal laws and pure implications; different types of effects of actions such as primitive and derived effect.

7.4.6 Reasoning on time intervals and events with duration

Event calculus is based on a point based representation of time. In [All83], Allen promotes the use of an interval representation of time. He argues that the notion of a time point should be replaced completely by the notion of a time interval. He introduces a calculus with thirteen possible relations between two different intervals I_1, I_2. These relations include relations such as: an interval I_1 is before I_2, I_1 meets I_2, I_1 overlaps with I_2, etc..

It is clear that reasoning on intervals is important. In many applications, there will arise questions like: does fluent P remain true or false during some specified time interval, does the value of some fluent change during some interval, what is the maximal time interval in which a fluent is true or false? No matter what

are Allen's feelings against time points, interval reasoning can easily be integrated with the point based representation of time of the event calculus[1]. The classical view on an interval as a pair of points can be defined without problem in event calculus, Allen's thirteen relations can be defined in terms of the three relations between time points ($t_1 \ll t_2, t_1 = t_2, t_2 \ll t_1$). New predicates $Holds\_during/2$ and $Not\_Holds\_during/2$ can be defined easily. Below follow variants of solutions which have already appeared elsewhere, e.g. in [KS92]:

$$Holds\_during(P,]T_1, T_2]) \text{ :- } Time(E), E \leq\leq T_1, Initiates(E, P), \\ \neg Clipped(E, P, T_2)$$
$$E \leq\leq E \text{ :-}$$
$$E \leq\leq T \text{ :- } E \ll T$$
$$Not\_Holds\_during(P,]T_1, T_2]) \text{ :- } \neg Holds(P, T_1), \neg Initiated\_during(P, [T_1, T_2[)$$
$$Not\_Holds\_during(P,]T_1, T_2]) \text{ :- } Time(E), E \ll T_1, Terminates(E, P), \\ \neg Initiated\_during(P, [T_1, T_2[)$$
$$Initiated\_during(P, [T_1, T_2[) \text{ :- } Time(E), In(T_1, E, T_2), Initiates(E, P)$$

A related problem is that of events with duration. In [KS86], it is argued that event calculus (based on events instead of time) is neutral with respect to the question whether events have duration or not. In our version this is not the case, because the notion of time is explicit and an event is associated to a unique time point. However, there is a simple and general way to introduce events with duration in terms of events without duration: split up each event in a starting and an ending event and introduce a fluent associated with the process of executing the event. For example, an event of closing a door can be split up in two events *start_close_door*, *end_close_door* and a fluent *closing_door*. These events are related by the following rules:

$$Initiates(E, closing\_door) \text{ :- } Act(E, start\_close\_door)$$
$$Terminates(E, closing\_door) \text{ :- } Act(E, end\_close\_door)$$
$$Succeeds(E) \text{ :- } Act(E, end\_close\_door), Holds\_at(closing\_door, E)$$

7.4.7 Concurrent Events

It is well-known from [KS86] that in contrast to situation calculus, event calculus allows to represent concurrent actions. Representing concurrent actions poses no *special* problems: it is in principle allowed that one time point is associated via $Act/2$ with distinct events. However, especially in the context of an undefined $\ll$ predicate, the issue of concurrent events must be considered with care, because unexpected and undesired solutions may be easily constructed. The problem whether two events *can* occur concurrently and what is their effect has no general solution

[1] The original event calculus [KS86] was based both on events and time intervals. The interest of interval reasoning was one of the authors' motivations for developing event calculus. In the later version, introduced by [Sha90], the notion of an interval was dropped again.

and domain dependent integrity constraints are needed to control concurrency. For example, it is clear that in a multiple robot problem, one robot can only be involved in one event at a given time. Distinct robots can perform concurrent actions, as long as they do not obstruct each other, e.g. two robots cannot pick up the same object at the same time.

An example illustrates the type of problems that can occur with concurrent events. Recall the light switch problem of section 7.2. A variant based on events is specified as follows:

Domain dependent clauses:
$Initiates(E, on)$:- $Act(E, Ev), Event\_Type(Ev, flip\_switch), \neg Holds\_at(on, E)$
$Terminates(E, on)$:- $Act(E, Ev), Event\_Type(Ev, flip\_switch),$
$Holds\_at(on, E)$
Problem specific information:
$Event\_Type(ev_1, flip\_switch)$:-
$Event\_Type(ev_2, flip\_switch)$:-
Integrity constraints:
$Time(start), Time(t_{end})$

There are two events ev_1, ev_2 of flipping a switch at unknown times. The predicates $Time/1$, $\ll$ and $Act/2$ are undefined. As before, the light at t_{end} should be off. Consider the following goal:

$\leftarrow Time(E1), Act(E1, ev_1), Time(E2), Act(E2, ev_2),$
$Holds\_at(on, t_{end}), E1 \ll t_{end}, E2 \ll t_{end}$

It queries whether it is possible that there exist time points $E1$, $E2$ on which ev_1, ev_2 occur, such that the light is *on* at t_{end}. The answer should be negative but SLDNFA-LO returns the solution:

$\Delta = \{Time(sk), Act(sk, ev_1), Act(sk, ev_2), start \ll sk \ll t_{end}\}$

The two distinct flip switch events occur at the same time.

The problem is not caused by an error of SLDNFA-LO but is due to the fact that the problem domain is under specified by the program. The current program expresses that two concurrent *flip_switch* events do not affect each others effects. Here this is not the case. In general we should express that an odd number of concurrent switch events changes the state of the light and an even number of concurrent events has no effect. A concise representation of these laws can be made in an extension of logic with sets and cardinality.

7.5 Declarative singularities

So far, the programs were interpreted under 3-valued completion semantics. From chapter 4, it follows that 3-valued completion semantics gives a safe approximation of a program under justification semantics: the completion is entailed by the

justification semantics. In this section we have a closer look at possible semantical problems which occur under completion and justification semantics.

In a first example, there is a fluent p which is initially false and a context dependent nonsense action a which initiates p if p is already true:

$Initially(X)$:- $\Box$
$Initiates(E, p)$:- $Act(E, a), Holds\_at(p, E)$

This example abstracts realistic problems in which pairs or sequences of actions may occur which have a zero overall effect. For example, a robot can pick up some object and put it back on the same location. This pair of events has no net effect.

Predicates $Time/1, Act/2$ are abducible, $\ll$ is a linear order with type predicate $Time/1$. As one might expect, SLDNFA-LO goes into an infinite loop when trying to solve the goal $\leftarrow Holds(p, t_{end})$. The following set of a events are generated:

$$start \ll \ldots \ll e_{-n} \ll \ldots \ll e_{-1} \ll t_{end}$$

This infinite loop is not purely a problem of SLDNFA-LO but hides a singularity at the declarative level. From the declarative point of view, one would expect that since a cannot initiate p properly, $\neg Holds(p, t_{end})$ should be logically implied by the theory. This is not the case wrt to completion semantics: there exist a model of the completion in which $Holds(p, t_{end})$ is true, and which extends the infinite set constructed in the limit by SLDNFA-LO. Indeed, the following 2-valued Herbrand interpretation is a model:

$\{Time(start), Time(t_{end})\} \cup$
$\{Time(e_i), Act(e_i, a) \mid -\infty < i < 0\} \cup$
$\{start \ll e_i \mid -\infty < i < 0\} \cup$
$\{e_j \ll e_i \mid -\infty < j < i < 0\} \cup$
$\{e_i \ll t_{end} \mid -\infty < i < 0\} \cup$
$\{Holds\_at(p, t_{end})\} \cup$
$\{Holds\_at(p, e_i) \mid -\infty < i < 0\} \cup$
$\{Initiates(e_i, p) \mid -\infty < i < 0\}$

For example, a $Holds\_at(p, e_i)$ atom has the following direct justification:

$$\{Time(e_{i-1}), e_{i-1} \ll e_i, Initiates(e_{i-1}, p), \neg Clipped(e_{i-1}, p, e_i)\}$$

An atom $Initiates(e_i, p)$ has a direct justification:

$$\{Act(e_i, a), Holds\_at(p, e_i)\}$$

From the procedural point of view, the existence of this model of the completion has the important consequence that no proof procedure which is sound wrt completion semantics, can ever succeed on the goal $\leftarrow \neg Holds\_at(p, t\_end)$.

It is easy to see that the above interpretation is not a justified model: the atom $Holds\_at(p, t_{end})$ has only justifications with positive loops. For this example, it can be shown that $\neg Holds\_at(p, t_{end})$ is implied under justification semantics. However, also for justification semantics, problematic situations can arise. A desirable property of an event calculus would be that it is overall consistent (see chapter 4). Unfortunately, it is easy to find examples with three-valued justified models. Consider the light switch problem. The unique fluent is *on*, the action is *flip_switch*:

$$Initiates(E, on) \text{ :- } Act(E, flip\_switch), \neg Holds\_at(on, E)$$
$$Terminates(E, on) \text{ :- } Act(E, flip\_switch), Holds\_at(on, E)$$

Note that two successive *flip_switch* events have no net effect. Hence, as in the previous example, there exists a model in which a looping problem occurs. The difference is that whereas in the previous example, positive looping occurs, here looping over negation occurs, due to the fact that $Initiates/2$ depends on $\neg Holds/2$. The following 3-valued Herbrand interpretation is a justified model:

$$\{Time(start)^{\mathbf{t}}, Time(t_{end})^{\mathbf{t}}\} \cup$$
$$\{Time(e_i)^{\mathbf{t}}, Act(e_i, flip\_switch)^{\mathbf{t}} \mid -\infty < i < 0\} \cup$$
$$\{(start \ll e_i)^{\mathbf{t}} \mid -\infty < i < 0\} \cup$$
$$\{(e_j \ll e_i)^{\mathbf{t}} \mid -\infty < j < i < 0\} \cup$$
$$\{(e_i \ll t_{end})^{\mathbf{t}} \mid -\infty < i < 0\} \cup$$
$$\{Holds\_at(on, t_{end})^{\mathbf{u}}\} \cup$$
$$\{Holds\_at(on, e_i)^{\mathbf{u}} \mid -\infty < i < 0\} \cup$$
$$\{Initiates(e_i, on)^{\mathbf{u}} \mid -\infty < i < 0\} \cup$$
$$\{Terminates(e_i, on)^{\mathbf{u}} \mid -\infty < i < 0\}$$

Figure 7.7 gives a weak justification for $Holds\_at(on, e_i)$. A few remarks are in order. A descendant node which points with two arrows to its ancestor represents a set of descendants. E.g. the node labelled with $In(x, e_{i-1}, e_i)$ $(x \neq e_{i-1})$ represents the set of descendants $\{In(x, e_{i-1}, e_i) \mid \forall x \in HU : x \neq e_{i-1}\}$. Note also that in the figure, the justifications for $In/3$ literals have not been written out.

The justification is clearly looping over negation since there are branches with $Holds\_at(on, e_i)$, $\neg Holds\_at(on, e_{i-1})$, $Holds\_at(on, e_{i-2})$, etc.. From this justification, weak justifications for the facts $\neg Holds\_at(on, e_i)$, $Initiates(e_i, on)$, $\neg Initiates(e_i, on)$, $Terminates(e_i, on)$, etc.. can easily be derived (and are implicit in the figure).

The problem with this model is due to its infinite sequence of actions, each of which depends on the previous in the sequence. How to avoid such sequences is a subject for future research.

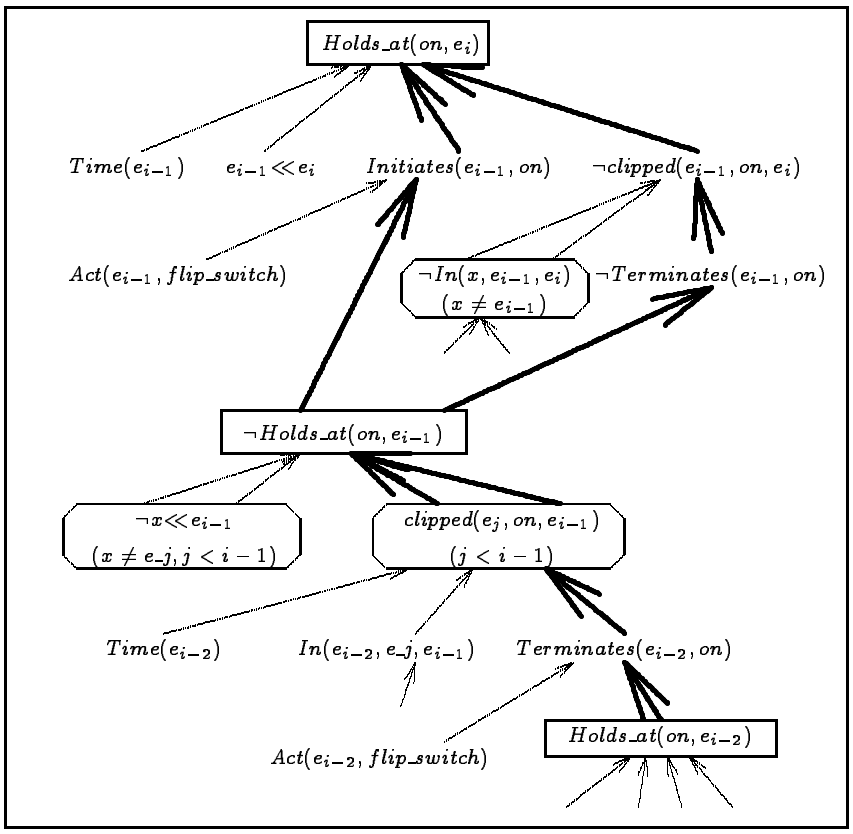

Figure 7.7: Justification for $Holds\_at(on, e_i)$

7.6 Planning with Incomplete Event Calculus

Planning problems can be represented in Event Calculus with undefined predicates $Time/1$, $Act/2$ and "$\ll$" [Esh88], [Sha89]. [Mis91b, Mis91a, MBD92] describes an implementation of such a planner and gives applications to a multiple robots block world problem, a multiple robots assembly problem and a room finishing problem. Below, a simple bread baking example is given and is gradually extended to illustrate solutions to several well-known difficulties in planning. SLDNFA-LO is used as a planning system. "$\ll$" is treated as a linear order with type predicate $Time/1$. Since we want minimal plans, the latter predicate is strongly abducible. $Act/2$ is abducible. In order to avoid problems with undesired concurrency, the

following constraint is added:

$$Act(E, Act1), Act(E, Act2) \rightarrow Act1 = Act2$$

The initial problem is to bake a bread from flour, yeast and water using the actions of kneading and baking. Kneading makes a dough of bread, baking turns a dough into bread. The problem can be represented by the following program and query:

$$\begin{aligned}
&Initiates(E, X) \text{ :- } Act(E, baking), Holds_at(dough(X), E)\\
&Terminates(E, dough(X)) \text{ :- } Act(E, baking)\\
&Initiates(E, dough(bread(E))) \text{ :- } Act(E, kneading(bread(E))),\\
&\qquad\qquad\qquad\qquad\qquad\qquad Holds_at(flour, E), Holds_at(water, E),\\
&\qquad\qquad\qquad\qquad\qquad\qquad Holds_at(yeast, E)\\
&Succeeds(E) \text{ :- } Act(E, baking)\\
&Succeeds(E) \text{ :- } Act(E, kneading(X))\\
&Initially(flour) \text{ :-}\\
&Initially(water) \text{ :-}\\
&Initially(yeast) \text{ :-}
\end{aligned}$$

The planner solves the query

$$\leftarrow Holds\_at(bread(X), t_{end})$$

by returning the solution:

$$\{Time(e1), Act(e1, baking), Time(e2), Act(e2, kneading(bread(e2))),\\ e2 < e1, e1 < t_{end} \quad \}$$

The baking event is a simple example of a *context dependent initiating event*: baking produces a bread only if a dough was present. The planner reduces the initial goal to

$$\leftarrow \quad Holds\_at(dough(bread(X)), e1)$$

This goal is solved successfully by abducing the kneading event. In general, planning with context dependent initiating events may produce large trees of events, in which each event initiates properties necessary for its parent in the tree.

A more complex type of event is the *context dependent terminating event*: an event that terminates a property if some precondition holds. The complexity stems from the fact that these events switch positive to negative execution and vice versa: to maintain a desirable property (occurring in a positive goal) which risks to be terminated by a context dependent event, one can try to fail the precondition of the terminating event. Vice versa, an undesirable property (occurring in a negative goal) may be removed by abducing a terminating event and solving its preconditions in a positive goal. To illustrate this, we extend the example as follows: we want the bread to be fresh; a dough is initially fresh; this property is terminated by baking if the oven is not warmed up in advance. Initially the oven is cold, the *warming_up* event terminates this property. The representation is:

$Initiates(E, fresh(X))$:- $Initiates(E, dough(X))$
$terminates(E, fresh(X))$:- $Act(E, baking)$,
$Holds\_at(dough(X), E), Holds\_at(oven\_cold, E)$
$Terminates(E, cold\_oven)$:- $Act(E, warming\_up)$
$Succeeds(E)$:- $Act(E, warming\_up)$
$Initially(cold\_oven)$:-

To maintain the freshness of the bread in the goal

$$\leftarrow Holds\_at(bread(X), t_{end}), Holds\_at(fresh(bread(X)), t_{end})$$

the solver tries to fail the precondition $\leftarrow Holds\_at(oven\_cold, e1)$. This leads to a negative goal $\leftarrow \neg Clipped(start, cold\_oven, e1)$, which is switched to a positive goal $\leftarrow Clipped(start, cold\_oven, e1)$ and, via the rule for termination, to the abduction of a $warming\_up$ event: $Time(e3)$, $Act(e3, warming\_up)$, $e3 < e1$. In general, due to context terminating events, positive and negative goals can alternate in any depth, and the abduction of events can occur in a positive goal at any depth.

Observe also that, in the previous plan, the order of the kneading and the warming_up events were not fixed. Another example showing this feature of the planner is in the extension for baking cake:

$initiates(E, dough(cake(E)))$:-
$Act(E, kneading(cake(E))), Holds\_at(flour, E),$
$Holds\_at(eggs, E), Holds\_at(sugar, E)$
$Initially(eggs)$:-
$Initially(sugar)$:-

$\leftarrow Holds\_at(bread(X), t_{end}), Holds\_at(cake(Y), t_{end})$

Three plans are generated by the system: one plan in which first the bread is kneaded and baked and then the cake, a second in which the order is reversed and a third in which the same baking event bakes both bread and cake. The latter plan leaves the order of the two independent kneading events undetermined. The kneading events become dependent if we add the additional requirement that a kneading event should happen with a clean kneading machine and kneading makes the machine dirty. The representation is:

$Succeeds(E)$:- $Act(E, kneading(X)), Holds\_at(clean, E)$
$Terminates(E, clean)$:- $Act(E, kneading(X))$
$Initially(clean)$:-

Observe that now the situation contains a problem similar to the light switching problem. In other approaches [Esh88], [Sha89]), [Mis91a], the third solution is still considered as correct, since by negation as failure they assume no time relation

between the kneading events. Our planner behaves correctly and terminates with failure.

The planning system provides other advanced features. Event calculus allows to represent derived properties in terms of primitive properties; integrity constraints can be added to the program; indeterminate actions can be represented. An interesting extension of classical planning is to drop the requirement that the initial situation must be known. For example, when we add the following clauses, the planner will generate not only the plan but also the necessary ingredients (*Ingredient* is abductive):

$Initially(X)$:- $Ingredient(X)$
$false$:- $Ingredient(X), \neg Base\_ingredient(X)$
$Base\_ingredient(flour)$:-
$Base\_ingredient(water)$:-
$Base\_ingredient(yeast)$:-
...

Our experiences with the planner have highlighted the need for an intelligent control strategy. We are currently implementing techniques from logic programming such as iterative deepening and intelligent control. Other techniques such as loop detection and intelligent backtracking could be helpful. In the abductive planner described in [Mis91a], several of these advanced control features have been implemented. In addition, the system offers the opportunity to implement domain dependent heuristics for avoiding infinite loops and unnecessary backtracking.

7.7 Discussion

Other features have been added to event calculus. [Eva90] extends event calculus with *event granularity*. He distinguishes between different levels of events. A high level event may be composed from other more primitive events. A simple example is the event of constructing a car, which is composed of events of constructing the engine, the bodywork and an assembly event. Each of these events may on turn be composed from more primitive events. A potential advantage of this different granularities occurs when it is possible to make abstraction of the fine grained actions involved in some process. In addition, [Eva90] introduces *time granularity*: time is also split up in different levels. In more coarse grained levels, time is more "discrete": e.g. on a high time level, the event of constructing a car may be instantaneous, while on more fine-grained levels, this event has a duration. The idea of time granularity was further refined in [Maï92]. A problem with these approaches is that it is unclear how the advantage of granularity could be exploited procedurally. The authors do not spend attention to procedural aspects such as in which situation a procedure can make safely abstraction of deeper levels of

granularity and when not. It is unclear to us whether these ideas have sufficient maturity to incorporate them in a working procedure like SLDNFA.

[KS92] used event calculus for emulating an object oriented database. One of the main problems with integrating object orientation in logic is the problem of dealing with change. [KS92] solves this problem elegantly in event calculus, by applying the technique proposed already in [KS86]: a historical database is maintained in which all updates are explicitly represented as events. In addition it is shown how different object oriented operations such as state change, creation and deletion of objects, dynamic change of class membership can be implemented.

So far, we specified time as a linear order. Though this suffices for many applications, it is a correct but rather coarse grained view. In many applications, time should be viewed as isomorphic with the real numbers, with operations like $\ll$, $+$, $\times$, $-$ and $/$. From the declarative point of view, numerical time is obtained by substituting the theory of real numbers for the theory of linear order. Note that $\ll$ in the real numbers is a linear order, such that it is correct to view numerical time as an instance of linear order time. Models with numerical time satisfy the theory of linear time.

With numerical time, a broad class of new problems can be represented. A classical "process control" example illustrate this. An initially empty tun must be filled with a volume v of some liquid. A tab allows to fill the tun at a fixed flow rate r. The type of solution that is expected here is of the form:

$$\Delta = \{Act(0, open\_tab), Act(v/r, close\_tab)\}$$

On the procedural level, the constraint solver for linear order must be replaced by a constraint solver for the real numbers. Many interesting problems could already be solved using a constraint solver based on the simplex algorithm.

A problem with applications of the above type and in many applications in which numerical time is important is that they involve *continuous change*. We considered so far only *instantaneous change*. Shanahan proposes an extension of event calculus to model continuous change [Sha90]. He introduces a predicate $Trajectory(N, T_1, V, T_2)$ with the following intended interpretation:

> there is a continuous change of type N which started at time T_1 and is continuing at least until time T_2 and the value at time T_2 is V.

Here N denotes a type of the continuous change, comparable to the notion of action as an event type. In this example $Trajectory/4$ is defined by:

$$Trajectory(filling, T_1, V, T_2) \text{ :- } Holds\_at(volume(V_1), T_1), \\ V = V_1 + r \times (T_2 - T_1)$$

The extended inertia axiom for trajectories is the following:

$$Holds\_at(P, T) \text{ :- } Time(E), E \ll T, Initiates(E, Q), \\ \neg Clipped(E, Q, T), Trajectory(Q, E, P, T)$$

There are events which start a trajectory and other events which terminate a trajectory. Here:

$Initiates(E, filling)$:- $Act(E, open\_tab)$
$Terminates(E, filling)$:- $Act(E, close\_tab)$

Additional axioms are needed when a continuous change *autoterminates*: i.e. when a continuous change causes its own termination. This happens for example when the tab overflows.

Shanahan's solution requires complete knowledge on the continuous change: the value at a given time is expressed via a numerical function. However, in many practical situations, one has incomplete knowledge about the continuous change. For example, the flow rate of a tab may be unknown or variable. Despite this incomplete knowledge, common sense reasoning infers that if a tab is opened and not closed then eventually there will be an overflow. We are currently investigating how Shanahan's approach can be generalised with incomplete knowledge.

7.8 Summary

At the declarative level, we have investigated event calculus as a declarative formalism for representing temporal knowledge. Special attention was spend on the laws of time. We have argued that the theory of linear order is of fundamental importance in event calculus. The theory should either be subsumed by the program, or should be added as a theory of integrity constraints. A number of classical benchmarks in temporal reasoning were solved correctly and elegantly. The grace of event calculus as a declarative formalism for temporal knowledge definitely equals, perhaps even surpasses that of situation calculus. Compared with situation calculus, event calculus seems to provide more elegant solutions for dealing with ramification, concurrency, events with duration, numerical time with applications in e.g. process control, continuous change, granularity, etc.. An interesting phenomenon observed in section 7.4.5 was the relationship between ":-" and causality. This relationship should be investigated further. It could possibly become one of the foundations of a future methodology for representing general temporal knowledge in event calculus. At a higher level, the experiments in this chapter can be seen as a second successful application of incomplete logic programming for the declarative representation of incomplete knowledge.

At the procedural level, our main contribution was to extend SLDNFA to SLDNFA-LO with a constraint solver for the theory of linear order. Recall that the problem of solving planning problems was one of the main motivations for developing SLDNFA. Interestingly, the resulting procedure generates partial plans: depending on the computation rule, the order of events is left unspecified when

the actions do not interfere. We showed how SLDNFA-LO can be applied for planning and for general temporal reasoning such as prediction, ambiguous prediction, postdiction and ambiguous postdiction.

Chapter 8

Conclusion

In our view on problem solving in a declarative language, a fundamental distinction should be drawn between the declarative representation of knowledge and the reasoning on the knowledge. The declarative part of problem solving is the description of the knowledge on the problem domain in a purely descriptive logic theory, without explicitly or implicitly coding the problem to be solved or the algorithm to be used for solving it. The correctness of the theory can in principle be evaluated on the basis of its model semantics: there must be some isomorphism between the possible states of the problem domain according to the users knowledge and the models of the problem description. This kind of analysis was done for example in chapter 7 in the context of event calculus, where we used model theoretic arguments to show that the theory of linear order is fundamental in event calculus. In practice, investigation of the models of a theory is often an impossible task. Therefore, one must resort to more informal ways of evaluating the correctness of the theory, by considering the *declarative reading* of the logic theory. This is a translation of the logic expressions using the user's *intended interpretation* to a natural language description of the problem domain. An essential property of a semantics of a logic is therefore that it provides a natural declarative reading.

The reasoning on the knowledge is the computational part of problem solving. On the problem domain, some specific problem must be solved. Different types of problems must be solved by different types of procedures. A problem can be classified under different computational paradigms. Some important classes are listed below:

- deduction: determine whether a formula is true in the problem domain
- abduction: find an explanation for some observation
- model generation: find a possible state of the problem which satisfies a set of constraints

- consistency proving: determine whether a certain observation is *possible* in the problem domain
- deductive database updating: given some database, update it with some new information
- model updating: given some model of a specification and some new constraint, find an update of the model such that the original constraints and the additional constraint are satisfied.
- etc...

In the thesis, we have made contributions on the declarative and the computational level. On the declarative level, we have studied the semantics of logic programs as sets of constructive definitions. We have argued that well-founded semantics and its extension, justification semantics, provided the most pure formalisation of the constructive definition view. We showed that the notion of constructive definition can be seen as an extension of the well-known concept of inductive definition. This gives logic programs a natural declarative reading.

Along the line of this study, a number of other important issues were considered. The current dominant view on model semantics in logic programming is that a model represents a *knowledge state*: a model describes what atoms are known true, what atoms are known false and what atoms are unknown. In contrast, we take a classical view on a model and see it as an abstraction of a possible state of the problem domain. This is exactly the role of a model in classical First Order Logic. A possible state semantics imposes a view on some fundamental issues in semantics, such as negation and incomplete knowledge. The negation operator is used to represent complementary concepts. Incomplete knowledge is represented by incomplete theories which have essentially different models. Having incomplete knowledge on the entities of the problem domain, is formalised by having models in which the domain is not the Herbrand universe. This was our motivation to reintroduce non-Herbrand interpretations in logic programming. Two applications in the temporal domain have shown how incomplete knowledge can be represented in incomplete logic programming.

In a possible state semantics, it makes no sense to interpret the third truth value by *unknown*. When a fact is unknown, this is reflected by having models in which the fact is true and others in which the fact is false. We proposed to interpret **u** as *locally inconsistent*. This interpretation is in accordance with the constructive definition view: we showed that in justification semantics (but also in stationary and well-founded semantics), **u** is only assigned to facts which cannot be interpreted consistently under the constructive definition view. This interpretation of **u** enables graceful degradation of the theory in the presence of inconsistent definitions. The abrupt collapse of two-valued semantics in classical FOL in case

of inconsistency, has been considered by many as a serious problem of classical logic. The introduction of local inconsistency copes elegantly with this problem.

Last but not least, our efforts for giving logic programs a FOL kind of semantics have allowed to integrate FOL into logic programming.

On the computational level, we developed new procedures for general model generation in first order logic with equality and for abduction in incomplete logic programs. Relationships between different computational paradigms were shown: between abduction and model generation, deduction, satisfiability proving and database updating.

We summarise the more technical contributions of our work, which were presented in chapters 3 to 7. In chapter 3, we have developed a model generator NMGE for FOL with equality. The procedure is an extension of the well-known model generator Satchmo with special techniques for dealing efficiently with equality. The intuition behind our approach is simple: sets of equality facts generated during model generation, are contracted to a compact representation which represents not only the set but also all its (possibly infinite) logic consequences under the underlying equality theory. The contraction is performed by transforming the set of equality facts in a *complete term rewriting system* and *normalising* the generated non-equality facts wrt to this term rewriting system. To obtain the desired generality, it was necessary to extend existing concepts of Term Rewriting.

We have illustrated the potential of the procedure for executing declarative specifications in FOL. A remarkable duality was found between the procedural semantics of abduction in incomplete definite logic programs and NMGE model generation in the only-if part of the programs.

Chapter 4 develops the semantical foundation of incomplete logic programming as a declarative formalism for representing incomplete knowledge. Here we defend the interpretation of logic programs as sets of constructive definitions. We investigated how current most popular semantics support this definition view. The leitmotif in this study was the notion of justification: the mathematical object which describes how the truth value of some fact in a model is justified by other facts. We argued that well-founded semantics and its extension to non-Herbrand interpretations, justification semantics are the only semantics in the study in which the justification for a positive fact does not contain the fact, i.e. the truth of a positive fact is not build on itself. This makes these semantics the best formalisation of the constructive definition view.

This work presents the 3-valued completion semantics as a safe approximation of the meaning of a theory in any other semantics in the framework. This means that models wrt any of the semantics are models wrt 3-valued completion semantics. A logic consequence of a program wrt 3-valued completion semantics is a logic consequence wrt to any semantics. In practice, a deductive and abductive procedure which is sound wrt completion semantics is sound wrt any other semantics

in the framework.

In chapter 5, we developed SLDNFA, an abductive procedure for the incomplete logic program formalism. SLDNFA is the first procedure which does not flounder on non-ground abductive atoms. We showed that for different application fields, different abductive procedures may be necessary, satisfying different completeness results. To cope with this problem, suitable extensions of SLDNFA were developed. This resulted in a (still simple) family of abductive procedures, SLDNFA^{o}_{+}, in which a number of parameters can be set to tune the procedure to the application under consideration.

Though SLDNFA was developed for abduction, the procedure turns out to be useful also for other computational paradigms. We showed how to use it soundly for deduction, satisfiability proving and database updating. So far the incomplete logic program formalism was associated uniquely with abduction as procedural paradigm. The possibility of reasoning with other computational paradigms, together with the declarative view on (incomplete) logic programs as (incomplete) constructive definitions turns the formalism into a full-fledged declarative logic.

In the two last chapters 6 and 7, we presented two different experiments in the declarative representation of (incomplete) knowledge and the use of SLDNFA for several forms of reasoning. Both experiments are in the context of temporal domains. In the first, we successfully translated $\mathcal{A}$ domain descriptions with incomplete knowledge on the initial situation to incomplete situation calculus with undefined predicate *Initially*/1. The main goal of this experiment was to compare incomplete logic programming with extended logic programming, another extension of logic programming which has been advocated for representing incomplete knowledge. Our transformation proved to be superior to Gelfond and Lifschitz's transformation to extended logic programming.

The second experiment in temporal reasoning was on (incomplete) event calculus. Whereas so far event calculus was mostly described in a procedural way, we investigated it as a declarative formalism. The grace of event calculus as a declarative formalism for temporal knowledge definitely equals that of situation calculus. Event calculus provides elegant solutions for dealing with ramification, concurrency, events with duration, numerical time with applications in e.g. process control, continuous change, granularity, etc.. An important contribution is the investigation of time as a linear order, and, at the procedural level, the extension of SLDNFA with a constraint solver for the theory of linear order. The resulting procedure is -to the best of our knowledge- the only procedure which generates correct partial plans. The procedure cannot only be used for planning but also for abduction, deduction and satisfiability proving in the context of general temporal reasoning.

Bibliography

[AB90] K.R. Apt and M. Bezem. Acyclic programs. In *Proc. of the International Conference on Logic Programming*, pages 579–597. MIT press, 1990.

[ABW88] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

[Acz77] P. Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland Publishing Company, 1977.

[AE82] K. R. Apt and M.H. Van Emden. Contributions to the Theory of Declarative Knowledge. *Journal of the ACM*, 29(3):841–862, 1982.

[All83] J.F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[AN78] H. Andreka and I. Nemeti. The Generalized Completeness of Horn Predicate Logic as a Programming Language. *Acta Cybernetica*, 4:3–10, 1978.

[Bak89] A.B. Baker. A simple solution to the Yale shooting problem. In *Proc. of the International Conference on Knowledge Representation and Reasoning*, pages 11–20, 1989.

[Bak91] A.B. Baker. Nonmonotonic Reasoning in the Framework of the Situation Calculus. *Artifical Intelligence*, 49:5–23, 1991.

[BLMM92] A. Brogi, E. Lamma, P. Mancarella, and P. Mello. Normal Logic Programs as Open Positive Programs. In K.R. Apt, editor, *Proc. of the International Joint Conference and Symposium on Logic Programming*, 1992.

[Bon92] P.A. Bonatti. Autoepistemic Logics as a Unifying Framework for the Semantics of Logic Programs. In K.R. Apt, editor, *Proc. of the International Joint Conference and Symposium on Logic Programming*, 1992.

[Bry90] F. Bry. Intensional Updates: Abduction via Deduction. In *Proc. of the International Conference on Logic Programming*, pages 561–575, 1990.

[CEP92] P.T. Cox, E.Knill, and T. Pietrzykowski. Abduction in Logic Programming with Equality. In *Proc. of International Conference on Fifth Generation Computer Systems*, pages 539–545, 1992.

[Chu36] A. Church. A note on the entscheidungsproblem. *JSL*, 1:40–41, 1936.

[CL89] L. Cavedon and J.W. Lloyd. A completeness theorem for sldnf resolution. *Journal of Logic Programming*, 7:177–191, 1989.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[CM85] E. Charniak and D. McDermott. *Introduction to Artifical Intelligence.* Addison-Wesley, 1985.

[CP86] P.T. Cox and T. Pietrzykowski. Causes for events: their computation and application. In *Proc. of the 8th International Conference on Automated Deduction*, 1986.

[CTT91] L. Console, D. Theseider Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[DD92a] M. Denecker and D. De Schreye. On the duality of abduction and model generation. In *Proc. of the International Conference on Fifth Generation Computer Systems*, 1992.

[DD92b] M. Denecker and D. De Schreye. SLDNFA; an abductive procedure for normal abductive programs. In K.R. Apt, editor, *Proc. of the International Joint Conference and Symposium on Logic Programming*, 1992.

[DD93a] M. Denecker and D. De Schreye. Justification semantics: a unifying framework for the semantics of logic programs. In *Proc. of the Logic Programming and Nonmonotonic Reasoning Workshop*, 1993.

[DD93b] M. Denecker and D. De Schreye. Representing incomplete knowledge in abductive logic programming. In *Proc. of the International Symposium on Logic Programming*, 1993.

[DD94] M. Denecker and D. De Schreye. On the Duality of Abduction and Model Generation in a Framework for Model Generation with Equality. Journal of Theoretical Computer Science; tentatively scheduled for Volume 122, 1994.

[Dec89] H. Decker. The range form of deductive databases and queries, or: how to avoid floundering. In J. Rettie and K. Leidmair, editors, *Proc. 5th ÖGAI*. Springer Verlag, 1989.

[Den92] J. Denef. Mathematical logic. course notes 2nd licence Mathematics, K.U.Leuven, 1992.

[Der87] N. Dershowitz. Completion and its applications. In *Proc. of the CREAS*, 1987.

[Dix92] J. Dix. Classifying Semantics of Disjunctive Logic Programs. In K.R. Apt, editor, *Proc. of the International Joint Conference and Symposium on Logic Programming*, 1992.

[DJ89] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, vol.B*, chapter 15. North-Holland, 1989.

[DMB92] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *Proc. of the European Conference on Artificial Intelligence*, 1992.

[Dun91] P.M. Dung. Negations as hypotheses: an abductive foundation for Logic Programming. In *Proc. of the International Conference on Logic Programming*, 1991.

[Dun92] P.M. Dung. Acyclic disjunctive logic programs with abductive procedure as proof procedure. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 555–561, 1992.

[Dun93] P.M. Dung. Representing Actions in Logic Programming and its Applications in Database Updates. In *Proc. of the International Conference on Logic Programming*, 1993.

[EK89] K. Eshghi and R.A. Kowalski. Abduction compared with negation as failure. In *Proc. of the International Conference on Logic Programming*. MIT-press, 1989.

[Esh88] K. Eshghi. Abductive planning with event calculus. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of the International Conference on Logic Programming*, 1988.

[Eva89] C. Evans. Negation as failure as an approach to the Hanks and McDermott problem. In *Proc. of the second International Symposium on Artificial Intelligence*, 1989.

[Eva90] C. Evans. The Macro-Event Calculus: Representing Temporal Granularity. In *Proc. of PRICAI, Tokyo*, 1990.

[Fag90] F. Fages. A New Fixpoint Semantis for General Logic Programs Compared with the Well-Founded and the Stable Model Semantics. In D.H.D. Warren and P. Szeredi, editors, *Proc. of the International Conference on Logic Programming*, page 443. MIT press, 1990.

[Fef70] S. Feferman. Formal theories for transfinite iterations of generalised inductive definitions and some subsystems of analysis. In A. Kino, J. Myhill, and R.E. Vesley, editors, *Intuitionism and Proof theory*, pages 303–326. North Holland, 1970.

[FG85] J.J. Finger and M.R. Genesereth. Residue: a deductive approach to design synthesis. Technical Report STAN-CS-85-1035, Department of Computere Science, Stanford University, 1985.

[FHKF92] M. Fujita, R. Hasegawa, M. Koshimura, and H. Fujita. Model Generation Theorem Provers on a Parallel Inference Machine. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 357–375. ICOT, Institute for New Generation Computer Technology, 1992.

[Fit85] M. Fitting. A Kripke-Kleene Semantics for Logic Programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[Fre67] G. Frege. Begriffsschrift, a Formula Language Modelled upon that of Arithmetic, for Pure Thought. In J. van Heijenoort, editor, *From Frege to Gödel: A source Book in Mathematical Logic, 1879-1931*, chapter 6, pages 1–82. Harvard University Press, 1967.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the International Joint Conference and Symposium on Logic Programming*, pages 1070–1080. IEEE, 1988.

[GL90a] M. Gelfond and V. Lifschitz. Logic Programs with Classical Negation. In D.H.D. Warren and P. Szeredi, editors, *Proc. of the 7th International Conference on Logic Programming 90*, page 579. MIT press, 1990.

[GL90b] A. Guessoum and J.W. Lloyd. Updating knowledge bases ii. Technical Report TR-90-13, Department of Computer Science, University of Bristol, 1990.

[GL92] M. Gelfond and V. Lifschitz. Describing Action and Change by Logic Programs. In *Proc. of the 9th Int. Joint Conf. and Symp. on Logic Programming*, 1992.

[HL89] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In H. D. Abramson and M. H. Rogers, editors, *Proceedings of Meta88*, pages 23–51. MIT Press, 1989.

[HM87] S. Hanks and D. McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, 1987.

[Hue80] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, 1980.

[K. 31] K. Gödel. Ueber formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatsh. Math. Phys.*, 37:349–360, 1931.

[KB70] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*, volume 1 of *Bibliotheca Mathematica*. Van Nostrand & Wolters-Noordhoff/North-Holland, Princeton, NJ & Groningen/Amsterdam, 1952.

[KM90a] A.C. Kakas and P. Mancarella. Database updates through abduction. In *Proc. of the 16th Very large Database Conference*, pages 650–661, 1990.

[KM90b] A.C. Kakas and P. Mancarella. Generalised stable models: a semantics for abduction. In *Proc. of the European Conference on Artificial Intelligence*, 1990.

[KM90c] A.C. Kakas and P. Mancarella. Stable Theories for Logic Programs. In *Proc. of the International Symposium on Logic Programming*, pages 85–100. The MIT-press, 1990.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *Proc. of IFIP 74*, pages 569–574. North-Holland, 1974.

[Kow79] R.A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–431, 1979.

[Kow90] R.A. Kowalski. Problems and promises of computational logic. In J.W. Lloyd, editor, *Proc. of the 1st Symposium on Computational Logic*, pages 1–36. Springer-Verlag, 1990.

[Kow91] R.A. Kowalski. Logic programming in artificial intelligence. In *Proc. of the IJCAI*, 1991.

[Kow92] R.A. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 1992.

[KS86] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(4):319–340, 1986.

[KS92] F. Kesim and M. Sergot. On the evolution of objects in a logic programming framework. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 1052–1060, 1992.

[Kun89] K. Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4:231–245, 1989.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[LMM88] J.-L. Lassez, M.J. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.

[LT84] J.W. Lloyd and R.W. Topor. Making prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[Maï92] E. Maïm. Abduction and Constraint Logic Programming. In *Proc. of the European Conference on Artificial Intelligence.* Wiley&sons, 1992.

[MB87] R. Manthey and F. Bry. A hyperresolution-based proof procedure and its implementation in prolog. In *Proc. of the 11th German workshop on Artificial Intelligence*, pages 221–230. Geseke, 1987.

[MBD92] L.R. Missiaen, M. Bruynooghe, and M. Denecker. Abductive planning with event calculus. Internal report, Department of Computer Science, K.U.Leuven, 1992.

[McC80] J. McCarthy. Circumscription - a form of nonmonotonic reasoning. *Artifical Intelligence*, 13:89–116, 1980.

[McD82] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.

[MD92] B. Martens and D. De Schreye. A perfect Herbrand semantics for untyped vanilla meta-programming. In K.R. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 511–525, Washington, November 1992. MIT Press.

[Men72] E. Mendelson. *Introduction to Mathematical Logic.* D. Van Nostrand company, 1972.

[Met83] Y. Metivier. About the rewriting systems produced by the knuth-bendix completion algorithm. *Information Processing Letters*, 16:31–34, 1983.

[Mis91a] L.R. Missiaen. *Localized abductive planning with the event calculus.* PhD thesis, Department of Computer Science, K.U.Leuven, 1991.

[Mis91b] L.R. Missiaen. Localized abductive planning for robot assembly. In *Proceedings 1991 IEEE Conference on Robotics and Automation*, pages 605–610. IEEE Robotics and Automation Society, 1991991.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[MMCR92] A. Montanari, E. Maïm, E. Ciapessoni, and E. Ratto. Dealing with Time Granularity in the Event Calculus. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 702–712, 1992.

[MMR86] A. Martelli, C. Moiso, and C.F. Rossi. An Algorithm for Unification in Equational Theories. In *Proc. of the Symposium on Logic Programming*, pages 180–186, 1986.

[MR90] J. Minker and A. Rajasekar. A fixpoint semantics for disjunctive logic programs;. *Journal of Logic Programming*, 9:45–74, 1990.

[PAA91a] L.M. Pereira, J.N. Aparicio, and J.J. Alferes. Derivation Procedures for Extended Stable Models. In J. Mylopoulos and R. Reiter, editors, *Proc. of the IJCAI.* Morgan Kaufmann Publishers, Inc., 1991.

[PAA91b] L.M. Pereira, J.N. Aparicio, and J.J. Alferes. Hypothetical Reasoning with Well Founded Semantics. In B. Mayoh, editor, *Proc. of the 3th Scandinavian Conference on AI.* IOS Press, 1991.

[Pei55] C.S. Peirce. *Philosophical Writings of Peirce.* Dover Publications, New York, 1955.

[Poo88] D. Poole. A Logical Framework for Default Reasoning. *Artificial Intelligence*, 36:27–47, 1988.

[Pop73] H. Pople. On the mechanization of abductive logic. In *Proc. of the 3d IJCAI*, pages 147–152, 1973.

[Prz88] T.C. Przymusinski. On the semantics of Stratified Databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufman, 1988.

[Prz89] T.C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Prz90] T.C. Przymusinski. Extended Stable Semantics for Normal and Disjunctive Programs. In D.H.D. Warren and P. Szeredi, editors, *Proc. of the seventh international conference on logic programming*, pages 459–477. MIT press, 1990.

[Prz91] T.C. Przymusinski. Well-Founded Completions of Logic Programs. In Koichi Furukawa, editor, *Proc. of the International Conference on Logic Programming*, pages 726–741. MIT press, 1991.

[Ram88] A. Ramsay. *Formal Methods in Artifical Intelligence*. Cambridge University Press, 1988.

[Rei78a] R. Reiter. Deductive Question-Answering on Relational Data Bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 149–177. Plenum Press, New York, 1978.

[Rei78b] R. Reiter. On Closed World Data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[Rei80] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Rei91] H. Reichgelt. *Knowledge Representation: an AI Perspecitive*. Ablex Publishing Corporation, 1991.

[Rei92] R. Reiter. Formalizing Database Evolution in the Situation Calculus. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 600–609, 1992.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[San91] E. Sandewall. Features and Fluents. Technical Report LiTH-IDA-R-91-29, Institutionen for datavetenskap, Linköping University, 1991. Preliminary version of a forthcoming book.

[Sha89] M. Shanahan. Prediction is deduction but explanation is abduction. In *Proc. of the IJCAI89*, page 1055, 1989.

[Sha90] M. Shanahan. Representing continuous change in the event calculus. In *Proc. of the European Conference on Artificial Intelligence*, page 598, 1990.

[Sho67] J. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Mass., 1967.

[SI92] K. Satoh and N. Iwayama. A Query Evaluation method for Abductive Logic Programming. In K.R. Apt, editor, *Proc. of the International Joint Conference and Symposium on Logic Programming*, 1992.

[SK88] F. Sadri and R.A. Kowalksi. A Theorem-Proving Approach to Database Integrity. In J. Minker, editor, *Foundations of Deductive Database and Logic Programming*, pages 313–362. Morgan Kaufman Publishers, 1988.

[Sny89] W. Snyder. Efficient ground completion: An o(n log n) algorithm for generating reduced sets of ground rewrite rules equivalent to a set of ground equations. In *Proc. of the 3rd International Conference on Rewriting Techniques and Applications*, 1989.

[THT87] D.S. Touretzky, J.F. Horty, and R.H. Thomason. A Clash of Intuitions: The Current State of Nonmonotonic Multiple Inheritance Systems. In *Proc. of the IJCAI87*, 1987.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[vEK76] M. van Emden and R.A Kowalski. The semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 4(4):733–742, 1976.

[VRS91] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

Appendix A

Expressive power of the Extended Clause formalism.

We prove that for each first order logic theory $\mathcal{T}$ based on $\mathcal{L}$ there exists an elementary extension theory $\mathcal{T}'$, consisting of extended clauses and based on a language $\mathcal{L}$' which extends $\mathcal{L}$ by a finite set of predicate symbols. Recall from chapter 2 that this means that each model of $<\mathcal{L}, \mathcal{T}>$ can be extended to a model of $<\mathcal{L}', \mathcal{T}'>$ and vice versa, that the restriction of a model of $<\mathcal{L}', \mathcal{T}'>$ to the symbols of $\mathcal{L}$ is a model of $\mathcal{T}$. This implies that $\mathcal{T}'$ is a *conservative extension* of $\mathcal{T}$ [Sho67]. This form of equivalence is stronger than the form of equivalence which has been proven for a theory $\mathcal{T}$ and its clausal form $\mathcal{T}'$: $\mathcal{T}$ is consistent iff $\mathcal{T}'$ is consistent.

We use the following terminology. We denote the fact that F_c is a sub-formula of F by $F_c \leq F$, and that F_c is a strict sub-formula of F by $F_c < F$. The set of *components* of a formula F are defined as the set of maximal strict sub-formulas of F. A conjunction and disjunction have two components; negations, universal and existential formulas have one component, an atom has no component. For each sub-formula F_c in F, there exists a linear chain of formulas $F_c = F_0 < F_1 < \ldots < F_n = F$, where each F_i is a component of F_{i+1} (although our notation does not make this explicit, we are talking about occurrences of sub-formulas rather than of sub-formulas directly; this is to avoid problems in the case of a sub-formula with multiple occurrences). The set $\{F_1, \ldots, F_n\}$ is precisely the set of formulas F' such that $F_c < F' \leq F$. The depth of F_c in F is n. The depth of a formula is recursively defined as the maximum depth of its components augmented with one. F_c occurs in a positive context or occurs positively in F if the number of formulas G such that $F_c < \neg G \leq F$ is even. Otherwise F_c occurs in a negative context or occurs negatively in F.

We assume that in a closed formula F each variable occurs with precisely one

quantifier. When this is not the case, renaming is always possible. Further, we require that each formula contains only the connectors $\wedge, \vee$ and $\neg$ and moreover, that each negation in the formula has an atom as component. Each first order logic formula can be transformed to an equivalent formula which satisfies these conditions, using the first part of the transformation in algorithm 4.9.1.

In section 3.2, the notion of extended clause was defined. The lemma below gives another characterisation.

Lemma A.1 *A formula F is an extended clause iff it satisfies the following constraints:*

- *the component of a negation is an atom*
- *the components of a conjunction $G \leq F$ are atoms or conjunctions;*
- *the component of an existential formula $G \leq F$ is either an atom or a conjunction or an existential formula;*
- *the components of a disjunction $G \leq F$ are atoms, negations, conjunctions, existential formulas or disjunctions;*
- *a universal formula $G \leq F$ can have any type of component.*

Proof A sketch of the proof is given. An extended clause is a formula of the form:

$$\forall X_1 \ldots \forall X_f : \neg A_1 \vee \ldots \neg A_g \vee E_1 \vee \ldots \vee E_h$$

where E_i has the general form:

$$\exists Y_1, \ldots, Y_i : A_1 \wedge \ldots \wedge A_j$$

Each A_i denotes an atom and $f, g, h, i \geq 0; j > 0$ and $g + h > 0$.

It is straightforward that an extended clause satisfies the syntactical constraints of the lemma. Vice versa, assume that F satisfies the syntactic constraints of the lemma. A proof by induction on the depth of the formula F can be given. The idea is as follows: by the induction hypothesis the components of F are known to be extended clauses. Since F satisfies the constraints in the lemma, the type of F restricts the type of its components. A simple case analysis suffices to show that F must be an extended clause too. For example, let F be a disjunction $G_1 \vee G_2$. G_1, G_2 can be any formula except a universal formula and they are both extended clauses. Hence they are both of the form $\neg A_1 \vee \ldots \vee \neg A_g \vee E_1 \vee \ldots \vee E_h$. The disjunction of two formulas of this form is again of this form (strictly spoken, commutativity and associativity must be applied here). Hence F is an extended clause. □

Below, an algorithm is given which transforms any theory to an extended program. The transformation proceeds by iteratively replacing an unwanted sub-formula of a formula (i.e. a sub-formula of a type which is not allowed by the lemma) by an atom of a new predicate, and adding a formula which relates this new predicate and the replaced sub-formula.

Algorithm A.1 *Let $\mathcal{T}$ be a theory based on $\mathcal{L}$. The transformation algorithm is defined as follows. Initially, set $\mathcal{T}' = \mathcal{T}$. As long as $\mathcal{T}'$ contains unwanted formulas, the following transformation step is executed:*

- *Select F from $\mathcal{T}'$ such that F contains a sub-formula G with an unwanted component $F_c[\overline{X}]$.*
- *Choose a new predicate p/n.*
- *Define F' by replacing F_c in F by $p(\overline{X})$.*
- *If F_c is a universal formula then F_c is of the form $\forall \overline{Y} : F'_c$ with F'_c not a universal formula; define $F'' = \forall \overline{X}, \overline{Y} : \neg p(\overline{X}) \vee F'_c$*
  Otherwise, define $F'' = \forall \overline{X} : \neg p(\overline{X}) \vee F_c$
- *Define $\mathcal{T}' := \mathcal{T} \setminus \{F\} \cup \{F', F''\}$.*

Theorem A.1 *Let $\mathcal{T}$ be any first order theory. The transformation algorithm terminates always. It produces an extended program $\mathcal{T}'$ based on an extension $\mathcal{L}'$ of $\mathcal{L}$ with new predicate symbols. $<\mathcal{L}', \mathcal{T}'>$ is an elementary extension of $<\mathcal{L}, \mathcal{T}>$.*

Proof To see that the algorithm terminates, just check that each transformation step decreases the number of unwanted components with one.

That the resulting $\mathcal{T}'$ is an extended program is trivial since no formula in $\mathcal{T}'$ contains an unwanted component.

We should prove (a) that each model of $<\mathcal{L}, \mathcal{T}>$ can be extended to a model of $<\mathcal{L}', \mathcal{T}'>$ and (b) that the restriction of a model of $<\mathcal{L}', \mathcal{T}'>$ to the symbols of $\mathcal{L}$ is a model of $<\mathcal{L}, \mathcal{T}>$. For the proofs of (a) and (b), we simplify the situation: we first show that if $\mathcal{T}'$ is obtained by $\mathcal{T}$ by one transformation step then (a) and (b) hold. Then by induction, (a) and (b) hold for the complete transformation process.

Assume that the transformation step selects F from $\mathcal{T}'$. F contains a sub-formula with an unwanted component F_c with free variables $\overline{X}$. To prove (a), let M be a model of $<\mathcal{L}, \mathcal{T}>$. We extend M to M' by extending the truth function of M' in the following way: for each variable assignment V of the variables $\overline{X}$: $\mathcal{H}_{M'}(V(p(\overline{X})) = \mathcal{H}_M(V(F_c))$

By definition, $M' \models \forall \overline{X} : (p(\overline{X}) \leftrightarrow F_c)$. From this equivalence, it is easy to prove that $M' \models F' \wedge F''$. In the case that F_c is a universal formula, the proof relies on the equivalence:

$$(\forall \overline{X} : (p(\overline{X}) \rightarrow \forall \overline{Y} : F'_c)) \Leftrightarrow (\forall \overline{X}, \overline{Y} : p(\overline{X}) \rightarrow F'_c)$$

which holds because $\overline{X}$ and $\overline{Y}$ are disjunct.

To prove (b), we use Lemma A.2 which is formulated and proven below: assume that F contains a sub-formula $F_c[\overline{X}]$ in a positive context; let F' be obtained by replacing F_c by F'_c, a formula with the same free variables. Then $(\forall \overline{X} : (F'_c \rightarrow F_c)) \rightarrow (F' \rightarrow F)$ is a tautology.

Since $M' \models \forall \overline{X} : (p(X_1, \ldots, X_k) \rightarrow F_c)$ and $M' \models F'$, it follows that $M' \models F$.

□

The above transformation procedure is far from optimal in the sense that it often introduces a large number of new predicates. In general a better result will be obtained if $\mathcal{T}$ is first pre-processed by distributing existential quantifiers over disjunctions and universal quantifiers over conjunctions.

Lemma A.2 *Let $F[\overline{Y}]$ be a formula containing a sub-formula $F_c[\overline{X}]$. Let F' be the formula obtained by replacing F_c by F'_c. If F_c occurs in a positive context in F then $\forall \overline{X} : (F_c \rightarrow F'_c)$ implies $\forall \overline{Y} : (F \rightarrow F')$. If F_c occurs in a negative context then $\forall \overline{X} : (F_c \rightarrow F'_c)$ implies $\forall \overline{Y} : (F' \rightarrow F)$.*

Proof The proof is by induction on the depth of F_c wrt F. The induction step is based on the following implications which can easily be proved for example in a model theoretical way:

$$\begin{aligned}
\forall X_1, \ldots, X_n : (F \rightarrow F') \quad \Rightarrow \\
&\forall X_1, \ldots, X_{n-1} : ((\forall X_n : F) \rightarrow (\forall X_n : F')) \\
&\forall X_1, \ldots, X_{n-1} : ((\exists X_n : F) \rightarrow (\exists X_n : F')) \\
&\forall X_1, \ldots, X_n : (F \wedge G \rightarrow F' \wedge G) \\
&\forall X_1, \ldots, X_n : (F \vee G \rightarrow F' \vee G) \\
&\forall X_1, \ldots, X_n : (\neg \mathrm{F} \leftarrow \neg F')
\end{aligned}$$

The case that F_c occurs at depth 0 in F (i.e. $F_c = F, F'_c = F'$) is trivial.

Assume that the lemma is proved for formulas in which F_c occurs at depth d. We prove the lemma for a formula F in which F_c occurs at depth $d+1$. F is either of the form $\exists X : F_1$, $\forall X : F_1$, $F_1 \wedge F_2$, $F_1 \vee F_2$ or $\neg F_1$.

If F_c occurs positively (negatively) at depth $d+1$ in F and F is not a negation $\neg F_1$ then F_c occurs positively (negatively) in F_i ($i = 1$ or $i = 2$) at depth d. By the induction hypothesis it holds that $\forall Y_1, \ldots, Y_k : F_i \rightarrow F'_i$ (or $\forall Y_1, \ldots, Y_k : F_i \leftarrow F'_i$ if F_c occurs negatively in F_i). Here F'_i is the formula

obtained by substituting F_c for F'_c in F_i. From the above implications, it follows directly that $\forall \overline{Y} : F \rightarrow F'$.

When F_c occurs negatively (positively) in $\neg F_1$, then F_c occurs positively (negatively) in F_1. Because of the induction hypothesis, it holds that $\forall Y_1, \ldots, Y_k : F_1 \rightarrow F'_1$ (or $\forall Y_1, \ldots, Y_k : F_1 \leftarrow F'_1$ if F_c occurs negatively in F_1). Because of the implication for the negation, the implication switches. So the lemma holds.

□

Appendix B

Mathematical foundation for Justifications.

(Open) justifications are complex objects: trees of facts with possibly infinite branching and infinite depth. In the paper, many proofs rely on two properties of justifications: that a monotonically increasing sequence of open justifications converges to an open justification and that open justifications can be concatenated on the leaves with other open justifications. In this appendix, we provide a mathematical foundation for open justifications and show that these properties hold. To do so, we abstract an open justification as a *tree* on some set D, where D corresponds to the set of all simple facts of some interpretation.

The intuition behind this formalisation is to map a tree on the set of all finite paths occurring in the tree and starting at the root. This gives a natural mapping between the nodes of the tree to the corresponding finite sequences which leads to the node.

Definition B.1 *A tree T on a set D, called the domain, is a set of finite sequences of elements of D, called paths, such that*

- *all sequences start with the same element of D, called the root.*
- *for all finite sequences of length d, all subsequences starting from the root occur in T.*

A node N at depth d can be defined as a finite sequence of length $d+1$ in T. A child N' of N is an extension of depth $d+1$. A leaf is a node which has no children. The label of a node is the last element of the sequence.

The set of finite sequences $\mathcal{F}$ of elements of D is a set. A tree is a subset of $\mathcal{F}$. The set of trees is a subset of the set of subsets of $\mathcal{F}$.

There is a natural partial order on trees: $T_1 \leq T_2$ means that T_2 extends T_1. $\leq$ can be defined in a trivial way:

Definition B.2 *Let T_1, T_2 be trees. We define $T_1 \leq T_2$ iff $T_1 \subseteq T_2$.*

This partial order satisfies the following proposition.

Proposition B.1 *For each monotonically increasing sequence of trees (T_i), there exists a LUB tree. This tree has the property that it contains all nodes which occur in one of the trees.*

Proof Consider the set $\bigcup(T_i)$. That this is the LUB of (T_i) wrt to $\leq$ is inherited directly from well known properties of $\subseteq$ and $\cup$. It is easy to proof that it is a tree. □

Next we define the concatenation of trees. Let l_1, l_2 be finite paths, by $l_1 + l_2$, we denote the sequence which is the concatenation of l_1 and l_2.

Definition B.3 *Let T be a tree. Let S be a subset of the leaves of T, and h a mapping from S to a set of trees such that for each N in S, $h(N)$ is a tree with the label of N as root. The concatenation of T and h is the tree $T \cup \{l_1 + l_2 \mid l_1 \in S$ and $l_2 \in h(l_1)\}$.*

Proposition B.2 *The concatenation of trees is a tree.*

The proof is easy, and is discarded.

Appendix C

Acyclic incomplete programs.

In [AB90] the notion of acyclic complete program is defined. We recall the basic concepts. They apply without change to incomplete logic programs.

Given is a complete or incomplete normal logic program P based on $\mathcal{L}$. P is acyclic iff there exists a level mapping $|.|$ for it. A level mapping $|.|$ is a mapping from $HB \cup \sim HB$ to $\mathbb{N}$ such that for each ground atom: $|A| = |\neg A|$ and for each ground instance $A \text{:-} L_1, \ldots, L_n$ of a normal clause of P: $|A| > |L_i|$ for each i.

Note that undefined predicates can always be assigned level 0, since they do not occur in the head of clauses.

A level mapping is extended to a mapping $\|.\|$ from all literals of $\mathcal{L}$ to $\mathbb{N}^\infty$, by:

$$\|L\| = max\{|\theta(L)| \mid \theta \text{ is a grounding substitution of } L\}$$

A literal L is called bounded if $\|L\|$ is finite.

Two properties proven in [AB90] are of interest below: for each clause of the form $A \text{:-} L_1, \ldots, L_n$ of P, each literal L and each substitution θ:

$$\begin{array}{ll} \|\theta(A)\| > \|\theta(L_i)\| & \text{if } \theta(A) \text{ is bounded} \\ \|\theta(L)\| < \|L\| & \text{if } L \text{ is bounded.} \end{array}$$

A new concept is that of a level mapping in an interpretation or pre-interpretation.

Definition C.1 *Let M be a pre-interpretation or (incomplete) interpretation of $\mathcal{L}$. We define $\|.\|_M$ as a mapping from the simple facts of M to $\mathbb{N}^\infty$. For each simple fact F we define:*

$$\|F\|_M = min\{\|L\| \mid L \textit{ is a literal and for some variable assignment } V: \tilde{M}(V(L)) = F\}$$

Because each subset of $\mathbb{N}^\infty$ contains a least element, we have that there exists a literal L_F and variable substitution V_F such that $\|F\|_M = \|L_F\|$ and $\tilde{M}(V_F(L_F)) = F$.

Lemma C.1 *(a) Let J_d be a direct justification of a bounded simple fact F. For each $F' \in J_d$, $\|F\|_M > \|F'\|_M$.*

(b) The depth of a justification of a bounded fact F is bound by $\|F\|_M + 1$.

Proof First assume that F is a positive fact. It suffices to prove that for each direct positive justification ($\mathcal{DPJ}$) $J_d = \{F_1, \ldots, F_n\}$, there exists an instance $\theta(A \text{ :- } L_1, \ldots, L_n)$ of a clause of P and a variable assignment V such that $\tilde{M}(V(\theta(A)) = F$, $\|\theta(A)\| = \|F\|_M$ and similarly for each $1 \leq i \leq n$: $\tilde{M}(V(\theta(L_i)) = F_i$, $\|\theta(L_i)\| = \|F_i\|_M$.

Since J_d is a $\mathcal{DPJ}$, there exists a ground domain instance $V'(A \text{ :- } L_1, \ldots, L_n)$ such that $\tilde{M}(V'(A)) = F$ and for $1 \leq i \leq n$: $\tilde{M}(V'(L_i)) = F_i$. Take a pair (L_F, V_F) such that $\|F\|_M = \|L_F\|$ and $\tilde{M}(V_F(L_F)) = F$. Analogously, for each $1 \leq i \leq n$ select for a pair (L_{F_i}, V_{F_i}) such that $\|F_i\|_M = \|L_{F_i}\|$ and $\tilde{M}(V_{F_i}(L_{F_i})) = F_i$. We may assume that the variables of L_F, L_{F_i} and $A \text{ :- } L_1, \ldots, L_n$ are disjunct. So, the set $V = V' \cup V_F \cup V_{F_1} \cup \ldots \cup V_{F_n}$ is a well-defined variable assignment.

Below, for any (domain) literal L, $\overline{t}_L$ denotes the (domain) terms appearing in L. So $L = p(\overline{t}_L)$ or $L = \neg p(\overline{t}_L)$. Note that for F, we have $\tilde{M}(V(\overline{t}_A)) = \overline{t}_F = \tilde{M}(V(\overline{t}_{L_F}))$. We have similar equations for F_i. We find that:

$$M \models V(\overline{t}_A = \overline{t}_{L_F} \wedge \overline{t}_{L_1} = \overline{t}_{L_{F_1}} \wedge \ldots \wedge \overline{t}_{L_n} = \overline{t}_{L_{F_n}})$$

Hence, the following closed formula is satisfied in M:

$$\exists(\overline{t}_A = \overline{t}_{L_F} \wedge \overline{t}_{L_1} = \overline{t}_{L_{F_1}} \wedge \ldots \wedge \overline{t}_{L_n} = \overline{t}_{L_{F_n}})$$

By theorem 5.4.2, there exists a unifier θ of these terms and $M \models V(\theta)$. As a consequence, $M \models V(\theta(\overline{t}_A)) = \overline{t}_F$ and for each $1 \leq i \leq n$, $M \models V(\theta(\overline{t}_{L_i})) = \overline{t}_{F_i}$. Because "=" is interpreted by the identity relation, we find:

$$\begin{array}{ll} \tilde{M}(V(\theta(A))) = F, & \text{and hence } \|F\|_M \leq \|\theta(A)\| \\ \tilde{M}(V(\theta(L_i))) = F_i, & \text{and hence } \|F_i\|_M \leq \|\theta(L_i)\| \end{array}$$

The equalities and identities below are now straightforward:

$$\begin{array}{l} \|\theta(A)\| = \|\theta(L_F)\| \leq \|L_F\| = \|F\|_M \leq \|\theta(A)\| \\ \|\theta(L_i)\| = \|\theta(L_{F_i})\| \leq \|L_{F_i}\| = \|F_i\|_M \leq \|\theta(L_i)\| \end{array}$$

Hence $\|\theta(A)\| = \|F\|_M$ and $\|\theta(L_i)\| = \|F_i\|_M$. Because $\|\theta(A)\| > \|\theta(L_i)\|$ we find $\|F\|_M > \|F_i\|_M$.

When F is a negative fact, a direct justification J_d is constructed by selecting facts from direct positive justifications from $\sim F$. It follows that for $F' \in J_d$, $\|F\|_M = \|\sim F\|_M > \|\sim F'\|_M = \|F'\|_M$.

Finally, it is clear that the length of a branch in a justification of a bounded fact F is bound by $\|F\|_M + 1$. □

A number of interesting properties hold about acyclic logic programs. These extend propositions in [AB90] for incomplete logic programs and justification semantics.

Proposition C.1 *(a) All the following semantics have the same Herbrand models: 2-valued and 3-valued (direct) (partial) justification semantics, 2-valued and 3-valued completion semantics [CTT91], generalised stable semantics [KM90b], generalised well-founded semantics [PAA91b].*

(b) The sets of implied ground literals of P + FEQ *under (direct) (partial) justification semantics are identical.*

(c) P + FEQ + SDCA *is overall consistent under justification semantics. Its semantics coincides with the generalised stable semantics [KM90b] and the generalised well-founded semantics [PAA91b].*

Unfortunately, the set of implied ground literals of $P + FEQ$ and $P + FEQ + SDCA$ may differ in general and hence Herbrand model based semantics gives other (more) implications than for example completion semantics. Assume $\mathcal{L}$ contains only $a/0$ as functor. Take $P =$

p :- $\neg q(X)$
$q(a)$:-

P is acyclic. In any Herbrand model wrt $(\mathcal{DJS})$ $(\mathcal{PJS})$ $(\mathcal{JS})$, p is false. But in each model in which more than one domain element occurs, p is true.

Acyclic programs with *FEQ* but without *SDCA* are not necessarily overall consistent. Consider the following complete program P based on a language with functors $s/1$ and a:

$$\{p(s(X)) \text{ :- } \neg p(X)\}$$

P is acyclic. Take the disjunct union of $\mathbb{N}$ and $\mathbb{Z}$, and interpret a by $0 \in \mathbb{N}$ and $s/1$ by the successor function on $\mathbb{N}$ and $\mathbb{Z}$. This pre-interpretation can be extended to a justified model M for which $\mathcal{H}_M(p(z)) = \mathbf{u}$ for each $z \in \mathbb{Z}$. $\mathcal{H}_M(p(n))$ is true for odd n and false for even n.

Proof Take any incomplete interpretation M_0 satisfying *FEQ*. It can be extended to a unique justified model M (theorem 4.3.4). In M, bounded simple facts have finite depth justifications. By an extension of the argument in proposition 4.3.3, we find that M is two-valued on all bounded facts and

each (directly) (partially) justified model M' extending M_0 assigns the same truth value to a bounded fact as M.

The proof of (a) continues as follows. When M_0 is a Herbrand incomplete interpretation, then all facts are bounded and hence each (directly) (partially) justified model coincides with the two-valued justified model extending M_0. By the theorems of chapter 4, the remaining semantics coincide also.

To finish (b), it suffices to see that each ground literal L is bound. Hence its truth value in any (directly) (partially) justified model corresponds with its truth value in a justified model. Hence the set of entailed ground literals of $P + FEQ$ is the same for all these semantics.

(c) follows from (a) and the equivalence theorems between semantics proven in chapter 4. □

For a restricted class of acyclic programs, the sets of entailed ground literals are identical under all semantics concidered in this thesis.

Definition C.2 *A head-restricted normal clause A :- $L_1, \ldots, L_n$ has the property that a variable appearing in the body appears in the head.*

The translation πD of an $\mathcal{A}$ domain D in chapter 6 is head-restricted. For this type of programs we have the following strong property:

Proposition C.2 *A theory $\mathcal{T}$ with a head-restricted acyclic logic program P and integrity constraints without variables has the property that the set of entailed ground atoms is equivalent wrt all the following semantics: (direct) (partial) justification semantics with* FEQ*, (partial) justification semantics with* FEQ *and* SDCA*, 2-valued and 3-valued completion semantics [CTT91], generalised stable semantics [KM90b], the generalised well-founded semantics [PAA91b].*

Proof First assume that $\mathcal{T}$ has no integrity constraints. By proposition C.1, a directly justified Herbrand model of an acyclic logic program is a model wrt to any of the other semantics. Also, each model of any type of semantics is a directly justified model; this follows from the theorems in chapter 4. Below we will prove that for a head-restricted acyclic program, for any directly justified model M, there exists a Herbrand directly justified model M_h such that for each ground literal L, $\mathcal{H}_M(L) = \mathcal{H}_{M_h}(L)$. This suffices for the theorem. Indeed, assume that some literal L is true in all models of a specific semantics, then also in all directly justified Herbrand models. Vice versa, assume that L is satisfied in all directly justified Herbrand models. Then for any model M of a specific semantics, M is a directly justified model, hence there exists a corresponding directly justified Herbrand model M_h, and hence L is true in M_h and M.

So let M be a directly justified model of $P + FEQ$. Note that if M satisfies FEQ, then $\tilde{M}$ defines a mapping from HU into the domain of M. We denote the image of HU under $\tilde{M}$ as D_h. $\tilde{M}$ is a one-to-one correspondence from HU to D_h. We define a Herbrand interpretation M_h as follows: for each $F \in HB$: $\mathcal{H}_{M_h}(F) = \mathcal{H}_M(F)$. Clearly, for each ground literal L, $\mathcal{H}_{M_h}(L) = \mathcal{H}_M(L)$. It remains to be shown that M_h is a directly justified model of P. This is done by showing that for any positive atom A, $\tilde{M}$ is a value preserving one-to-one correspondence between the direct positive justifications of A and of $\tilde{M}(A)$.

Take any $\mathcal{DPJ}$ J_d of A in M_h. There is a clause B :- $L_1, \ldots, L_n$ and a grounding substitution θ such that $\theta(B) = A$ and $J_d = \{\theta(L_1), \ldots, \theta(L_n)\}$. Consider the variable assignment $V = \{X/\tilde{M}(\theta(X)) \mid X \in dom(\theta)\}$. It is easy to see that $J'_d = \{\tilde{M}(V(L_1)), \ldots, \tilde{M}(V(L_n))\}$ is a $\mathcal{DPJ}$ of $\tilde{M}(A)$. Vice versa let J_d be any direct positive justification of $\tilde{M}(A)$. There exists a clause B :- $L_1, \ldots, L_n$ and a variable assignment V such that $\tilde{M}(V(B)) = \tilde{M}(A)$ and $J_d = \{\tilde{M}(V(L_1)), \ldots, \tilde{M}(V(L_n))\}$. Note that each variable in V occurs in B, since P is head-restricted. There exists a substitution θ of the variables of B such that $\theta(B) = A$. This can easily be proven using proposition 5.4.2. θ is a grounding substitution of all literals of the body. One easily verifies that $\tilde{M}(\theta(L_i)) = \tilde{M}(V(L_i))$. Hence the $\mathcal{DPJ}$ $\{\theta(L_1), \ldots, \theta(L_n)\}$ of A in M_h has the same value as J_d.

The case that $\mathcal{T}$ has integrity constraints without variables follows easily from the fact that the associated Herbrand interpretation M_h of a directly justified model M of $\mathcal{T}$ is not only a directly justified model of P but also of the integrity constraints of $\mathcal{T}$.

□