Víctor Pablos Ceruelo

NEGATIVE NON-GROUND QUERIES IN WELL FOUNDED SEMANTICS

Lisboa
2009

MCL

Víctor Pablos Ceruelo

2009

UNIVERSIDADE NOVA DE LISBOA
Facultade de Ciências e Tecnologia
Departamento de Informática

NEGATIVE NON-GROUND QUERIES IN WELL FOUNDED SEMANTICS

Por
Víctor Pablos Ceruelo

Dissertaçao apresentada na Facultade de Ciências e Tecnologia
da Universidade Nova de Lisboa para obtençao do grau de
Mestre em Computational Logic

Orientador: José Júlio Alves Alferes

Lisboa

2009

ABSTRACT

The existing implementations of Well Founded Semantics restrict or forbid the use of variables when using negative queries, something which is essential for using logic programming as a programming language.

We present a procedure to obtain results under the Well Founded Semantics that removes this constraint by combining two techniques: the transformation presented in [MMNMH08] to obtain from a program its dual and the derivation procedure presented in [PAP+91] to determine if a query belongs or not to the Well Founded Model of a program.

Some problems arise during their combination, mainly due to the original environment for which each one was designed: results obtained in the first one obey a variant of Kunen Semantics and non-ground programs are not allowed (or previously grounded) in the second one.

Most of these problems were solved by using abductive techniques, which lead us to observe that the existing implementations of abduction in logic programming disallow the use of variables.

The reason for that is the impossibility to evaluate non-ground queries, so it seemed interesting to develop an abductive framework making use of our negation system.

Both goals are achieved in this thesis: the capability of solving non-ground queries under Well Founded Semantics and the use of variables in abductive logic programming.

To my girlfriend,
Sara,
with love.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Kowalski and Colmerauer's choice of the elements of first-order logic supported in logic programming was influenced by the availability of implementation techniques and efficiency considerations. Among those important aspects not included from the beginning we can mention *evaluable functions*, *negation* and *higher order features*. All of them have revealed themselves important for the expressiveness of logic programming as a programming language. Here we propose an implementation for Well Founded Semantics (WFS, see Sec. 2.5) that solves some pending problems with *negation*: floundering when trying to solve non-ground negative queries

Negation in logic has been widely studied and its declarative semantics has been defined in multiple ways (see Chapter 2 for an overview). The development of the operational procedures for negation has been influenced by the necessity of using logic programs as programs (coding algorithms), while the development of the declarative semantics has been influenced by the purpose of using logic programs as a knowledge representation framework. These two different concerns lead to two different approaches to the meaning of logic programs, that we call respectively proof-theoretic approach and model-theoretic approach.

As a result of the different concerns in proof-theoretic and model-theoretic approaches, while most of the implementations of the former lack a correct management of contradictory, inconsistent or incomplete information, most of the model-theoretic ones do not consider the existence of variables in the programs or restrict their use. So, the first ones disallow the use of logic programming for knowledge representation and the second ones hinder the use of logic programming as a programming language.

Implementation examples of proof-theoretic methods are the (unsound) negation as failure rule, and the sound (but incomplete) delay technique of the language Gödel [HL94], Nu-Prolog [Nai86] (having the risk of floundering), the constructive negation of ECLiPSe [ECL] (which was announced in earlier versions but has been removed from recent releases due to implementation errors) or Constructive Intensional Negation [MMNMH08]. The ones of model-theoretic methods are ASP [Gel07], the XSB Prolog system [XSB], the DLV system for disjunctive datalog with constraints, true negation and queries [DLV, LPF$^+$06], the Stable Model Semantics system SMOD-

ELS [NSS00], the SLG system [CW93, SW94] and how to make it portable [RC94], tabled evaluation with delaying [CW96], or the derivation procedures for extended stable models [PAP+91].

Although some authors argue that programs with contradictory, inconsistent or incomplete information are erroneous and not useful, the fact is that when representing knowledge we can not get rid of them. The following program is an example of this.

Program 1.0.1

```
work ← ¬tired.                                    1
sleep ← ¬work.                                    2
tired ← ¬sleep.                                   3
angry ← work, ¬paid.                              4
paid ← .                                          5
```

This program naturally encodes the knowledge that: we work if we are not tired (1), if we do not work then we sleep (2), we get tired if we do not sleep (3), when we work and we are not paid we get angry (4), and we are always paid (5).

If we try to determine if we are paid it is clear that we are, but we can not determine if we work, we sleep, we are tired or we are angry. When evaluating the logic program the result is almost the same: we get an infinite loop when evaluating the queries *work*, *sleep*, *angry* or *tired*.

Depending on the point of view, we can describe our knowledge about these propositions as incomplete or perhaps even confusing. The model-theoretic methods focus in assigning a truth value (*undefined*, *false* or even *true*) to a proposition suffering from this problems. Their proposal is to associate a model (see Def. 2.1.19) to the program and answer the queries directed to the program in accordance to this model. The model is the *meaning of the program*, or its *declarative semantics*. For example, the *Well Founded Model* (WFM, see Def. 2.5.2) for program 1.0.1 is $\{$ *paid* $\}$, so that the atom *paid* belongs to the model and is thus true. Consequently, its negation, $\neg$*paid* , is false. Moreover, neither *work*, *sleep*, *angry*, *tired* nor their negations belong to our model and they are assigned the *undefined* truth value.

The definitions of the model-theoretic methods usually require that, in presence of variables, programs must be grounded before evaluating the model, and this grounding process might obtain an infinite ground program. This can be observed in the example program 1.0.2, where the Herbrand Universe (see Def. 2.1.9) is the infinite representation of the natural numbers: $\mathcal{U} = (\ 0,\ s(0),\ s(s(0)),\ \ldots\ )$.

Program 1.0.2 (Peano numbers)

```
natural(0).                                       1
natural(s(X)) ← natural(X).                       2
                                                  3
even(0).                                          4
even(s(X)) ← odd(X).                              5
odd(s(X)) ← even(X).                              6
```

As we usually tend to program by using variables and functor symbols to build the data structures of our programs (see Def. 2.1.2), instead of computing the Herbrand Universe to determine the model (which is unfeasible for a machine), the implementations rely on top-down methods for evaluating whether the query belongs or not to the model (see chapter 3). While for positive programs these methods works perfectly, when dealing with negative non-ground queries they do not work adequately: as they make use of negation as failure they suffer from floundering. We first introduce negation as failure and just after it the floundering problem.

Negation as (finite) failure is the most common negation mechanism in logic programming. It is a meta-inference-rule allowing one to prove the negation of a ground goal, when the proof of the corresponding positive goal finitely fails. So, to prove $\neg a$ in the following program we just have to prove that a finitely fails. As in this case a has a proof, $\neg a$ is not proven. On the contrary, b has no proof, and so $\neg b$ can be proved.

Program 1.0.3

```
a ← c.                                                    1
b ← d.                                                    2
c.                                                        3
```

While negation as failure works perfectly for non-ground queries, when dealing with variables it produces unexpected results. In the following example, to prove $\neg a(1)$ (which is again a ground query) we need to prove that $a(1)$ finitely fails, and from that result negation as failure allows us to prove $\neg a(1)$, which is the correct result. But to prove $\neg a(X)$ we need to prove that $a(X)$ finitely fails. As $a(X)$ has a proof for $X = 2$, it does not finitely fail and we get no proof for $\neg a(X)$. So, while we expected a proof with the substitution $X = 1$ (or, more generally, with $X \neq 2$), what we get is "no".

Program 1.0.4

```
a(X) ← r(X).                                              1
r(2).                                                     2
```

Evaluation of queries $\neg a(1)$ and $\neg a(X)$ in program 1.0.4

```
?- ¬a(1).                    ?- ¬a(X).
   yes                          no
```

The problem becomes even worse if we have free variables in the body of one (or more) rules. In the following program we define that my (possible infinite) set of friends is composed by my direct friends, the direct friends of my friends and so on.

Program 1.0.5

```
friend(me, juan).                                          1
friend(me, pepe).                                          2
friend(juan, antonio).                                     3
friend(pepe, maria).                                       4
...                                                        5
friend(X, Z) ← friend(X, Y) , friend(Y, Z).                6
```

The ones which are not my friends are then the ones that are not direct friends of me or direct friends of one of my friends. If we make a positive non-ground (or ground) query, the top-down method works perfectly, and it obtains all my friends one by one.

Evaluation of query $friend(me,\ X)$ **in program 1.0.5**

```
?- friend(me, X).
   X = juan ? ;
   X = pepe ? ;
   X = antonio ? ;
   X = maria ? ;
   ...
```

Correct results are also guaranteed if we make a ground negative query. But if we make a negative non-ground query, before getting any result the first thing that the method will do is to determine which are my friends. As the set of my friends behaves like an infinite set, it is impossible to compute the whole set of friends.

The existing WFS implementations do not have a correct management of universally quantified variables and use basic methods that suffer from floundering[1] (Both examples 1.0.4 and 1.0.5 suffer from floundering, although the end of the computation in the second one presents an error, usually *OutOfMemory* exception).

The main objective of this work is to develop an implementation dealing correctly with non-ground queries under the Well Founded Semantics. Non ground queries in logic programming is solved in the negation system developed in
[MHMN00, MHMNH01, MH03, MMNMH08], where the authors incorporate negation in the Prolog system Ciao [Bue95]. They developed a system able to manage negation properly, combining different techniques and using static and dynamic tests to determine which method is the best in each case. They are able to select the built-in *negation as failure* (NAF, see Sec. 3.1) when a groundness analysis tells that every negative literal is ground at call time, select Finite Constructive Intensional Negation (an improved version of Constructive Intensional Negation that only works for finite domains) when the analysis tells that the program has no infinite answers, or using Constructive Intensional Negation (see Sec. 3.2) in the worst case.

Despite the huge amount of work done, the authors developed the system under a variant of Clark's Predicate Completion Semantics (see Sec. 2.2) that they suggest

[1] When the evaluation stops with no answer the problem is called floundering. This problem has been widely documented in the literature, and we refer the reader to [AB94] for a survey.

to be seen as a CLP version of Kunen's semantics (KS, see Sec. 3.2). The drawback of Kunen's Semantics, as other proof-theoretic methods, is that it does not work adequately in presence of contradictory, inconsistent or incomplete information.

In this thesis we generalize this work on Constructive Intensional Negation [MMNMH08] to answer (non-ground) queries under the Well Founded Semantics. This is done by combining the procedure used in [MMNMH08] with the one used in [PAP$^+$91].

Basically, we take from [MMNMH08] the methods to calculate the dual program, and the derivation procedure from [PAP$^+$91]. As the derivation procedure was not developed to deal with dual programs, it is modified to take into account the following characteristics that dual programs present:

1. the dual of a unification is a disequality. Since the predefined inequality predicate in logic programming is not suitable for dealing with variables (see Sec. 4.4), in the implementation we must use attributed variables (see Def. 4.4.1) to deal with them.

2. if a predicate in the original program has a free variable, its negation has this variable universally quantified (see Sec. 3.5). As logic programming does not have a predefined predicate to determine if a universal quantification holds or not, we must implement a method capable of determining that.

When solving these problems we found that the techniques used resemble abductive techniques (see chapter 5). In fact, abduction collects solutions and tests these solutions for the consistency of the result. Our implementation behaves like that in two parts:

1. inequalities implementation: we collect all the disequalities over a variable and finally we test if their conjunction is consistent or not (see Sec. 4.4).

2. universally quantified variables implementation: the results for the variable are collected and finally they are tested in order to find a tautology that guarantees the universal quantification to hold (see Secs. 3.5 and 4.5).

The extension of our work presented in chapter 5, abduction, comes indeed from its previous use to solve these problems: as abduction is used to solve them, the application of our negation system to abduction, in order to obtain explanations to negative hypotheses, seemed rather promising. The result from applying it is an abductive framework that benefits from this new implementation of negation, and that can make an unrestricted use of variables.

One extra feature that we considered mandatory for our negation system is that solutions must be obtained in a Prolog way, i.e. one by one. In [MMNMH08], instead, the solutions returned when making a query follow the structure

$$sol\_1 \vee sol\_2 \vee \ldots \vee sol\_n \ ?\ ;$$
$$no$$

Such a solution has the following drawbacks:

- results obtained can not be used from other Prolog programs without a previous conversion.
- one might be only interested in one solution (or a small subset of solutions), in which case it makes no sense to wait for computing the whole disjunction. This is specially important if our set of solutions has an exponential size, as may happen when the computation of the universal quantification is involved.
- if the number of solutions is infinite then computing them (or their disjunction) is unfeasible. While there is a solution, their method is not able to obtain any solution at all.

Desirably, this should be substituted by the Prolog way, so that instead of results joined by disjunction what we get is one result (disjunct) at a time, following the structure

sol_1 ? ;
sol_2 ? ;
...
sol_n ? ;
no

The implementation of the Well Founded Semantics that we finally present here, and that constitutes the main contribution of this thesis, has the following characteristics:

1. It can be applied to any kind of logic program in which no symbols are introduced to constraint the normal execution of Prolog, like cuts (the term "!"). So, no restriction exists on the use of variables in the program or in the queries.
2. Solutions are obtained one by one. Some solutions represent disequalities, and they use attributed variables for that purpose. Disequalities are only joined if they need to be fulfilled at the same time (and only by conjunction), so they represent only one valid solution.
3. The way inequalities between variables are tested is based on abductive techniques: inequality conditions are assumed in advance and stored in the attribute of the variables, being really tested when the variable is bound.
4. The way that the universal quantification is evaluated uses again abductive techniques: solutions to the universally quantified variables are collected and tested in order to build a tautology, so it can be determined if the universal quantification holds.
5. Answers are given according to the Well Founded Semantics.

The structure of this work is as follows: to understand the implementation issues a revision of the semantics is needed; it is in chapter 2. Chapter 3 presents

the existing problems in WFS implementations and a general idea of how they are solved by our implementation, which is explained in detail in chapter 4. After that, in chapter 5, we present the abductive framework that benefits from the negation system that we developed. Chapter 6 presents the conclusions of our work. The implementation, along with some illustrative examples, is available at "*https://babel.ls.fi.upm.es/software/intneg-wfs/*".

CHAPTER 2

SEMANTICS

According to [PP90], which we follow for this overview, the declarative semantics SEM(Π) of a logic program Π can be specified in various ways, among which the following two are most common. One that can be called proof-theoretic, associates with Π its first order completion COMP (Π) (e.g., COMP (Π) can be Π itself or the Clark predicate completion comp(Π) of Π). A formula V is said to be implied by the semantics SEM(Π) if and only if it is logically implied by the completion

$$\text{COMP } (\Pi) \models \text{V};$$

i.e., if V is satisfied in all 2-valued (Herbrand or not) models of COMP (Π).

Another method of defining the declarative semantics SEM(Π) of a program is model-theoretic. The semantics is determined by choosing a set MOD(Π) of intended models of Π (in particular, one intended model M_Π). For example, MOD(Π) can be the set of all minimal models of Π or the unique least model of Π. A formula V is said to be implied by the semantics SEM(Π) if and only if it is satisfied in all intended models:

$$\text{MOD}(\Pi) \models \text{V (in particular, } M_\Pi \models \text{V)}:$$

Observe, that the proof-theoretic approach can be viewed as a special case of the model-theoretic approach. Other approaches to defining the declarative semantics are possible, e.g., a combination of proof-theoretic and model-theoretic methods has been used in [Kun87, Fit85].

2.1 Preliminaries

In this section we introduce all the basic concepts and syntax that we use. In keeping with Prolog's convention, local variables begin with a capital letter; constants, functions and predicates begin with a lowercase letter.

Definition 2.1.1 (Alphabet)**.** *By an alphabet $\mathcal{A}$ of a first order language $\mathcal{L}$ we mean a (finite or countably infinite) set of constant, predicate and function symbols. In addition, any alphabet is assumed to contain a countably infinite set of variable symbols, connectives $(\vee, \wedge, \neg, \leftarrow)$, quantifiers $(\exists, \forall)$ and the usual punctuation symbols. Moreover, we assume that our language also contains propositions **t**, **u** and **f**, denoting the properties of being true, undefined or false.*

Definition 2.1.2 (Term, Functor)**.** *Terms are as customarily defined in logic:*

- *A variable or constant is a term.*
- *A function symbol (or functor) with terms as arguments is a term.*

Terms may also be viewed as data structures of the program, with function symbols serving as record names. The word ground is used as a synonym for "variable-free", in keeping with common practice. Often a constant is treated as a function symbol of arity zero.

Definition 2.1.3 (Predicate, Atomic Formulae, Atom)**.** *A predicate is a relation between terms, so the arguments of a predicate are terms. We use the same symbol, e.g., p, to refer both a predicate and its relation.*

If p is a predicate symbol of arity n ($n \geq 0$) and $t_1, ...t_n$ are terms then $p(t_1, ...t_n)$ is an atomic formulae or atom. Predicates of arity zero are also called propositions.

Definition 2.1.4 (Formulae)**.** *If F_1 and F_2 are formulas then so are*

- $\neg F_1$
- $F_1 \wedge F_2$
- $F_1 \vee F_2$
- $F_1 \leftarrow F_2$

If X is a variable and F is a formulae then $\exists X.F$ and $\forall X.F$ are formulas too. We say that X is existentially quantified in $\exists X.F$ and universally quantified in $\forall X.F$.

Definition 2.1.5 (Quantifier-free formulae)**.** *If $\mathcal{F}$ is a quantifier-free formulae, then by its ground instance we mean any ground formulae obtained from $\mathcal{F}$ by substituting ground terms from the Herbrand Universe $\mathcal{U}$ (see Def. 2.1.9) for all variables.*

Definition 2.1.6 (Universal closure of a formulae)**.** *For a given formulae $\mathcal{F}$ of a language $\mathcal{L}$ its universal closure or just closure $\forall\mathcal{F}$ is obtained by universally quantifying all variables in $\mathcal{F}$ which are not bound by any quantifier.*

Definition 2.1.7 (Expression, Sentence)**.** *An expression is a term or a formulae. A formulae with no free (unquantified) variables is a sentence.*

Definition 2.1.8 (Literal)**.** *A literal L is either an atom A or its negation $\neg A$. In the first form it is called positive literal and in the second one negative literal.*

Definition 2.1.9 (Herbrand Base, Herbrand Universe)**.** *The set of all ground atoms of an alphabet $\mathcal{A}$ is called the Herbrand Base $\mathcal{H}$ of $\mathcal{A}$. The set of all ground terms of $\mathcal{A}$ is called the Herbrand Universe $\mathcal{U}$ of $\mathcal{A}$.*

If the alphabet $\mathcal{A}$ contains a function symbol of positive arity, then the Herbrand Universe and Herbrand Base are countable infinite; otherwise they are finite.

Definition 2.1.10 (Logic Rule or Clause)**.** *A rule or a clause C is a formulae of the form*

$$H \leftarrow L_1 \wedge \ldots \wedge L_n$$

where H is an atom other than false and $\{ L_1, \ldots, L_n \}$ *is a possibly empty set of literals. The head or conclusion H is written on the left, and its subgoals (body) if any to the right of the symbol* $\leftarrow$*, which may be read "if". In some parts of this document we refer to the head by the left hand side (lhs) of the clause, and to the body by the right hand side (rhs) of the clause.*

Conforming to a standard convention, conjunction is replaced by commas and therefore clauses are simply written in the form

$$H \leftarrow L_1, \ldots, L_n$$

For example,

$$p(X) \leftarrow a(X), \neg\, b(X), \neg\, (c(X), d(X))$$

is a rule in which p(X) is the head, a(X) is a positive subgoal and b(X) and (c(X), d(X)) are negative subgoals. The head of a rule is always a positive literal.

Definition 2.1.11 (Logic Program (I))**.** *By a logic program* Π *we mean a finite set of logic rules or clauses. If* Π *is a logic program then, unless stated otherwise, we will assume that the alphabet* $\mathcal{A}$ *used to write* Π *consists precisely of all the constant, predicate and function symbols that explicitly appear in* Π *and thus* $\mathcal{A} = \mathcal{A}_\Pi$ *is completely determined*[1] *by the program* Π*. We can then talk about the first order language* $\mathcal{L} = \mathcal{L}_\Pi$ *of the program* Π *and the Herbrand base* $\mathcal{H} = \mathcal{H}_\Pi$ *of the program.*

Remark. Although some authors distinguish between general logic programs and normal logic programs (see [GRS91, Llo87]), we will use logic programs to refer all of them.

Definition 2.1.12 (Logic program (II))**.** *Programs are constructed from a signature* $\Sigma = \langle FS_\Sigma, PS_\Sigma \rangle$ *of function and predicate symbols. Provided a countable set of variables V the set* $Term(FS_\Sigma, V)$ *of terms is constructed in the usual way.*

Remark. This definition is equivalent to the previous one, but instead of defining the alphabet from the program it defines the program from the alphabet. It is used in the explanation of the Constructive Intensional Negation method (see Sec. 3.2).

Definition 2.1.13 (Horn Rule and Horn Logic Program)**.** *A horn rule is a general rule with no negative subgoals, and a Horn logic program is one only with Horn rules.*

Definition 2.1.14 ((Herbrand) constraint)**.** *A (Herbrand)* constraint *is a first-order formula where the only predicate symbol is the binary equality operator* $= /2$*. A formula* $\neg(t_1 = t_2)$ *will be abbreviated* $t_1 \neq t_2$*. The constants* $\mathbf{t}$ *(for true) and* $\mathbf{f}$ *(for false) will denote the neutral elements of conjunction and disjunction, respectively. A tuple* $(x_1, \ldots, x_n)$ *will be abbreviated by* $\overline{x}$*. The concatenation of tuples* $\overline{x}$ *and* $\overline{y}$ *is denoted* $\overline{x} \cdot \overline{y}$*.*

[1] If there are no constants in the program Π, then one is added to the alphabet.

Definition 2.1.15 ((constrained) Horn clause)**.** *A (constrained) Horn clause is a formula*

$$h(\overline{x}) \leftarrow b_1(\overline{y} \cdot \overline{z}),\ \ldots,\ b_n(\overline{y} \cdot \overline{z}) \parallel c(\overline{x} \cdot \overline{y})$$

where $\overline{x}$, $\overline{y}$ and $\overline{z}$ are tuples from disjoint sets of variables.[2] *The symbols "," and "$\parallel$" act here as aliases of logical conjunction (the second one is introduced for readability of the examples).*

Definition 2.1.16 ((constrained) program)**.** *A (constrained) program is:*

$$\begin{gathered} p(\overline{x}) \leftarrow B_1(\overline{y}_1 \cdot \overline{z}_1) \parallel c_1(\overline{x} \cdot \overline{y}_1) \\ \vdots \\ p(\overline{x}) \leftarrow B_m(\overline{y}_m \cdot \overline{z}_m) \parallel c_m(\overline{x} \cdot \overline{y}_m) \end{gathered}$$

The set of defining clauses for predicate symbol p in program Π is denoted $def_\Pi(p)$. Without loss of generality we have assumed that the left hand sides in $def_\Pi(p)$ are syntactically identical.

Definition 2.1.17 (Completed definition of a program)**.** *Assuming the normal form, let $def_\Pi(p) = \{p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i) \parallel c_i(\overline{x} \cdot \overline{y}_i) | i \in 1 \ldots m\}$. The* completed definition *of p, $cdef_\Pi(p)$ is defined as the formula*

$$\forall \overline{x}. \left[p(\overline{x}) \iff \bigvee_{i=1}^{m} \exists \overline{y}_i.\ c_i(\overline{x} \cdot \overline{y}_i) \wedge \exists \overline{z}_i . B_i(\overline{y}_i \cdot \overline{z}_i) \right]$$

Definition 2.1.18 (Interpretation)**.** *By a 3-valued Herbrand interpretation I of the language $\mathcal{L}$ we mean any pair $< T, F >$, where T and F are disjoint subsets of the Herbrand base $\mathcal{H}$. The set T contains all ground atoms true in I, the set F contains all ground atoms false in I and the truth value of the remaining atoms in $U = \mathcal{H} - (T \cup F)$ is undefined. We assume that in every interpretation I the proposition **t** is true, the proposition **f** is false and the proposition **u** is undefined. A 3-valued interpretation I is 2-valued if all ground atoms (except for the proposition **u**) are either true or false in I.*

An alternative way to represent an interpretation $I =< T, F >$ is $I = T \cup \{\neg L \mid L \in F\}$.

Definition 2.1.19 (Model)**.** *M is a model of a program Π if and only if the degree of the truth of the head of every clause is at least as high as the degree of truth of its body. To use this definition we need to introduce how to evaluate the degree of truth of the head and the body of the clauses.*

Any interpretation $I =< T, F >$ can be equivalently viewed as a function $I : \mathcal{H} \rightarrow \{ 0, \frac{1}{2}, 1 \}$, from the Herbrand base $\mathcal{H}$ to the 3-element set $\mathcal{V} = \{ 0, \frac{1}{2}, 1 \}$, defined by:

$$I(A) = \begin{cases} 0, & if A \in F \\ \frac{1}{2}, & if A \in U \\ 1, & if A \in T \end{cases}$$

[2] The notation $p(\overline{x})$ expresses that $Vars(p(\overline{x})) \in \overline{x}$, not that it is identical to $p(x_1, \ldots, x_n)$

We now extend the function (interpretation) $I : \mathcal{H} \rightarrow \mathcal{V}$ recursively to the truth valuation $\hat{I} : C \rightarrow \mathcal{V}$ defined on the set C of all closed formulae of the language.

[Prz89a] If I is an interpretation, then the truth valuation $\hat{I}$ corresponding to I is a function $\hat{I} : C \rightarrow \mathcal{V}$ from the set C of all (closed) formulae of the language to $\mathcal{V}$ recursively defined as follows:

- *If A is a ground atom, then $\hat{I}(A) = I(A)$.*
- *If S is a closed formulae then $\hat{I}(\neg S) = 1 - \hat{I}(S)$.*
- *If S and T are closed formulae, then*

$$\hat{I}(S \wedge T) = min\{\hat{I}(S), \hat{I}(T)\};$$
$$\hat{I}(S \vee T) = max\{\hat{I}(S), \hat{I}(T)\};$$
$$\hat{I}(T \leftarrow S) = \begin{cases} 1, & if \ \ \hat{I}(T) \geq \hat{I}(S) \\ 0, & otherwise \end{cases}$$

- *For any formulae S(X) with one unbounded variable X:*

$$\hat{I}(\forall X.S(X)) = min \ \{ \ \hat{I}(S(A)) : \ A \in \mathcal{H}_\Pi \ \};$$
$$\hat{I}(\exists X.S(X)) = max \ \{ \ \hat{I}(S(A)) : \ A \in \mathcal{H}_\Pi \ \};$$

where the maximum (resp. minimum) of an empty set is defined as 0 (resp. 1).

Once introduced how to evaluate the truth values of the head and the body of a clause we can expose that if our program is a set of rules of the form $A \leftarrow L_1, ..., L_m$, where $m \geq 0$, A's is an atom and L_i's are literals, then an (Herbrand) interpretation M is a model of a program Π if all the program clauses are true in M, i.e., if for every ground instance of a program clause we have

$$\hat{M}(A) \ \geq \ min \ \{ \ \hat{M}(L_i) : \ i \ \leq \ m \ \}$$

Thus, M is a model of a program Π if and only if the degree of the truth of the head of every clause is at least as high as the degree of truth of its body, as told before. By a ground instantiation of a logic program Π we mean the (possibly infinite) theory consisting of all ground instances of clauses from Π.

Corollary 2.1.1. *An (Herbrand) interpretation M is a model of a program Π if and only if it is a model of its ground instantiation.*

A program Π can have different Herbrand models, as can be seen in the following example. In fact, programs with negation may have several minimal Herbrand models.

Example 2.1.1 (Program with different Herbrand models)

```
p(1).                                    1
q(2).                                    2
q(X) ← p(X).                             3
```

model 1: {p(1), q(1), q(2)}
model 2: {p(1), p(2), q(1), q(2)}

Models are usually associated with programs to represent the "meaning of the program" or its "declarative semantics". The idea is that a declarative semantics for a class of logic programs can be defined by selecting, for each program Π in this class, one of its models to determine which answer to a given query is considered correct. For instance, a query without variables should be answered "yes" if it belongs to the model and no otherwise, and a query with variables should be answered "yes" if by substitution we found a ground version of the query that belongs to the model.

Ideally we should be able to select a canonical model for that purpose, but instead of that we have different approaches to select it, as Stable Models [GL88], Perfect Models [Prz88] and Well Founded Models (WFM) [GRS91, VGRS88]. Each one of them is linked with a semantics, which respectively are Stable Model Semantics (SMS) [GL88], Perfect Model semantics (PMS) [Prz88, ABW88, vG89], and Well Founded Semantics (WFS) [GRS91, VGRS88]. A survey on PMS and WFS is exposed in the following sections. We refer the reader to [GL88, PP90] for a survey on SMS.

Definition 2.1.20. *[Prz89a] If I and J are two interpretations then we say that $I \preccurlyeq J$ if*

$$I(A) \leq J(A) (or, equivalently, \hat{I}(A) \leq \hat{J}(A))$$

for any ground atom A. If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called minimal in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \preccurlyeq I$ and $J \neq I$. An interpretation I is called least in $\mathcal{I}$ if $I \preccurlyeq J$, for any other interpretation $J \in \mathcal{I}$. A model M of a theory R is called minimal (resp. least) if it is minimal (resp. least) among all models of R.

Proposition 2.1.1. *If $I =< T, F >$ and $I' =< T', F' >$ are two interpretations, then $I \preccurlyeq I'$ iff $T \in T'$ and $F \in F'$. In particular, for 2-valued interpretations, $I \preccurlyeq I'$ iff $I \in I'$.*

Thus I $\preccurlyeq$ I' if and only if I has no more true facts and no less false facts than I' . This means that minimal and least models of a theory R minimize the degree of truth of their atoms, by minimizing the set T of true atoms and maximizing the set F of false atoms F. In particular, the least interpretation in the set of all interpretations is given by I $=< \emptyset, \mathcal{H} >$.

As we mentioned above, [Fit85] considers a different ordering of truth values based on the degree of information rather than on the degree of truth. Under this ordering, the 'unknown' value is less than both values 'true' and 'false', with 'true' and 'false' being incompatible. This immediately leads to a different ordering between interpretations and to different notions of minimal and least models.

Definition 2.1.21. *[Fit85] If $I =< T, F >$ and I' $=< T', F' >$ are two interpretations, then we say that $I \preccurlyeq_F$ I' iff $T \subseteq$ T' and $F \subseteq$ F'. We call this ordering the F-ordering. If $\mathcal{I}$ is a collection of interpretations, then an interpretation $I \in \mathcal{I}$ is called F-minimal in $\mathcal{I}$ if there is no interpretation $J \in \mathcal{I}$ such that $J \preccurlyeq_F I$ and $J \neq I$. An interpretation I is called F-least in $\mathcal{I}$ if $I \preccurlyeq_F J$, for any other interpretation $J \in \mathcal{I}$. A model M of a theory R is called F-minimal (resp. F-least) if it is F-minimal (resp. F-least) among all models of R.*

In particular, the F-least interpretation in the set of all interpretations is given by I $=< \emptyset, \emptyset >$. The notions of F-minimal and F-least models are different from the notions of minimal and least models (see 2.1.20). While minimal and least models of a theory R minimize the degree of truth of their atoms, by minimizing the set T of true atoms and maximizing the set F of false atoms F, F-minimal and F-least models minimize the degree of information of their atoms, by jointly minimizing the sets T and F of atoms which are either true or false and thus maximizing the set U of unknown atoms. For example, the F-least model of the program $p \leftarrow p$ is obtained when p is undefined, while the least model of Π is obtained when p is false. As it will be seen in the sequel, this distinction reflects fundamental differences between the semantics based on Clark's completion and model-theoretic semantics, such as the least model semantics, perfect model semantics or well-founded semantics.

2.1.1 Fixed Points

Declarative semantics of logic programs is often defined using fixed points of some natural operators Ψ acting on ordered sets of interpretations. Suppose $\leq$ is an ordering on the set $\mathcal{I}$ of interpretations of a given language, $\mathcal{J}$ is a subset of $\mathcal{I}$ and Ψ is an operator $\Psi : \mathcal{I} \rightarrow \mathcal{I}$ on $\mathcal{I}$.

Definition 2.1.22. *The operator Ψ is called monotone if $I \leq J$ implies $\Psi(I) \leq \Psi(J)$, for any I, J $\in \mathcal{I}$. An interpretation $I \in \mathcal{I}$ is a fixed point of Ψ if $\Psi(I) = I$. By the least upper bound $\sum \mathcal{J}$ of $\mathcal{J}$ (resp. the greatest lower bound $\Pi\mathcal{J}$ of $\mathcal{J}$) we mean an interpretation $I \in \mathcal{I}$ such that $J \leq I$, for any $J \in \mathcal{J}$ and $J \leq J'$ for any other J' with this property (resp. $I \leq J$, for any $J \in \mathcal{J}$ and $J' \leq J$ for any other J' with this property). By the smallest interpretation (under the given ordering) we mean an interpretation I_0 such that $I_0 \leq I$, for any other interpretation I.*

Least fixed points of monotone operators Ψ are often generated by iterating the operator Ψ starting from the smallest interpretation I_0 and obtaining the (possibly transfinite) sequence:

$$\Psi^{\uparrow 0} = I_0;$$
$$\Psi^{\uparrow \alpha+1} = \Psi(\Psi^{\uparrow \alpha});$$
$$\Psi^{\uparrow \lambda} = \sum_{\alpha<\lambda} \Psi^{\uparrow \alpha};$$

for limit ordinals λ. Clearly, an iteration $\Psi^{\uparrow \alpha}$ is a fixed point of Ψ if and only if

$$\Psi^{\uparrow \alpha} = \Psi^{\uparrow \alpha+1}.$$

In the sequel we will consider two principal orderings among interpretations, namely the standard ordering $\preccurlyeq$ (see Def. 2.1.20) and the F-ordering $\preccurlyeq_F$ (see Def. 2.1.21). Operators acting on sets of interpretations ordered by the standard ordering, will be denoted by Ψ or Θ, while those acting on sets of interpretations ordered by the F-ordering, will be denoted by Φ or Ω. Recall that $I_0 =< \emptyset, \mathcal{H} >$ (resp. $I_0 =< \emptyset, \emptyset >$) is the smallest (resp. F-smallest) interpretation in the set of all interpretations ordered by $\preccurlyeq$ (resp. $\preccurlyeq_F$).

For a subset $\mathcal{J}$ of $\mathcal{I}$, we will denote by $\sum \mathcal{J}$ (resp. $\Pi\mathcal{J}$) the least upper bound (resp. the greatest lower bound) of $\mathcal{J}$ with respect to $\preccurlyeq$. Similarly, we will denote by $\sum_F \mathcal{J}$

(resp. $\Pi_F\mathcal{J}$) the least upper bound (resp. the greatest lower bound) of $\mathcal{J}$ with respect to $\preccurlyeq_F$.

Observe, that if $\mathcal{J} = \{J_s : s \in S\}$, with $J_s =< T_s; F_s >$, then:

$$\begin{aligned}
\textstyle\sum\mathcal{J} &=< \bigcup_{s\in S} T_s, \bigcap_{s\in S} F_s >;\\
\Pi\mathcal{J} &=< \bigcup_{s\in S} T_s, \bigcap_{s\in S} F_s >;\\
\textstyle\sum_F\mathcal{J} &=< \bigcup_{s\in S} T_s, \bigcup_{s\in S} F_s >;\\
\Pi_F\mathcal{J} &=< \bigcap_{s\in S} T_s, \bigcap_{s\in S} F_s > .
\end{aligned}$$

Although $\sum\mathcal{J}$, $\Pi\mathcal{J}$ and $\Pi_F\mathcal{J}$ are always well-defined interpretations, $\sum_F \mathcal{J} =< T, F >$ may not be an interpretation, because the sets T and F may not be disjoint. However, $\sum_F\mathcal{J}$ is always an interpretation, provided that $\mathcal{J}$ is an F-directed set of interpretations, i.e., such that for any J, J' $\in \mathcal{J}$ there is a J" $\in \mathcal{J}$ satisfying J $\preccurlyeq_F$ J" and J' $\preccurlyeq_F$ J".

2.2 Clark's Predicate Completion Semantics

The most commonly used declarative semantics of logic programs, although less popular in the context of knowledge representation, is based on the so called Clark predicate completion comp(Π) of a logic program Π [Cla78, Llo87]. Clark's completion of Π is obtained by first rewriting every clause in Π of the form:

$$\mathrm{q(K_1, ..., K_n)} \leftarrow \mathrm{L_1}, ..., \mathrm{L_m},$$

where q is a predicate symbol and $\mathrm{K_1}$, ..., $\mathrm{K_n}$ are terms containing variables $\mathrm{X_1}$, ..., $\mathrm{X_k}$, as a clause

$$\mathrm{q(T_1, ..., T_n)} \leftarrow \mathrm{V},$$

where $\mathrm{T_i}$'s are variables,

$$\mathrm{V} = \exists \mathrm{X_1}, ..., \mathrm{X_k}\ (\mathrm{T_1} = \mathrm{K_1} \wedge ... \ \mathrm{T_n} = \mathrm{K_n} \wedge \mathrm{L_1} \wedge ... \wedge \mathrm{L_m})$$

and then replacing, for every predicate symbol q in the alphabet, the (possibly empty[3]) set of all clauses

$$\begin{aligned}
\mathrm{q(T_1, ..., T_n)} &\leftarrow \mathrm{V_1}\\
&...\\
\mathrm{q(T_1, ..., T_n)} &\leftarrow \mathrm{V_s}
\end{aligned}$$

with q appearing in the head, by a single universally quantified logical equivalence

$$\mathrm{q(T1, ..., Tn)} \leftrightarrow \mathrm{V_1} \vee ... \vee \mathrm{V_s}.$$

[3]If there are no clauses involving the head $\mathrm{q(T_1, ..., T_n)}$, then the corresponding disjunction is empty and thus always false. The resulting completion contains therefore a universal negation of $\mathrm{q(T_1, ..., T_n)}$.

Finally, the obtained theory is augmented by the so called Clark's Equality Axioms (see Def. 2.2.1), which include unique names axioms and axioms for equality. These axioms are essential when considering non-Herbrand models of Clark's completion.

Definition 2.2.1 (Clark's Equational Theory (CET))**.** *The so called Clark's Equality Axioms [Kun87] are:*

CET1 $X = X$;

CET2 $X = Y \Rightarrow Y = X$;

CET3 $X = Y \wedge Y = Z \Rightarrow X = Z$;

CET4 $X_1 = Y_1 \wedge \ldots \wedge X_m = Y_m \Rightarrow f(X_1, \ldots, X_m) = f(Y_1, \ldots, Ym)$, *for any function f;*

CET5 $X_1 = Y_1 \wedge \ldots \wedge X_m = Y_m \Rightarrow (\ p(X_1, \ldots, X_m) \Rightarrow p(Y_1, \ldots, Ym) \ )$, *for predicate p;*

CET6 $f(X_1, \ldots, X_m) \neq g(Y_1, \ldots, Y_n)$, *for any two different function symbols f and g;*

CET7 $f(X_1, \ldots, X_m) = f(Y_1, \ldots, Y_m) \Rightarrow X_1 = Y_1 \wedge \ldots \wedge X_m = Y_m$, *for any function f;*

CET8 $t[X] \neq X$, *for any term* $t[X]$ *different from* X, *but containing* X.

Remark. The first five axioms describe the usual equality axioms and the remaining three axioms are called unique names axioms or freeness axioms. The significance of these axioms to logic programming is widely recognized [Llo87, Kun87]. The equality axioms (CET1) - (CET5) ensure that we can always assume that the equality predicate "=" is interpreted as identity in all models. Consequently, in order to satisfy the CET axioms, we just have to restrict ourselves to those models in which the equality predicate - when interpreted as identity - satisfies the unique names axioms (CET6) - (CET8).

Clark's approach is mathematically elegant and founded on a natural idea that in common discourse we often tend to use 'if' statements, when we really mean 'iff' statements. For example, we may use program 2.2.1 to describe natural numbers.

Program 2.2.1 (Peano numbers)

```
natural_number (0).                                          1
natural_number (succ(X)) ← natural_number (X).               2
```

The theory in program 2.2.1 is rather weak. It does not even imply that, say, Mickey Mouse is not a natural number. This is because, what we really have in mind is

$$\text{natural number (T)} \leftrightarrow \exists X\ (T = 0 \vee (T = \text{succ(X)} \vee \text{natural\_number(X)}))$$

which is in fact Clark's completion of program 2.2.1 and it indeed implies

$$\neg\text{natural\_number(MickeyMouse)}.$$

Unfortunately, Clark's predicate completion semantics has some serious drawbacks. One of them is the fact that Clark's completion is often inconsistent, i.e., it may not have any 2-valued (Herbrand or not) models, in which case Clark's semantics is undefined. For example, Clark's completion of the program $p \leftarrow \neg p$ is $p \leftrightarrow \neg p$, which is inconsistent. The situation can be even worse, e.g., Clark's completion of the program 2.2.2 is program 2.2.3,

Program 2.2.2

```
p ← ¬q, ¬p                                                    1
```

Program 2.2.3

```
p ← ¬q, ¬p                                                    1
¬q                                                            2
```

which is inconsistent. However, after adding to program 2.2.2 a 'meaningless' clause q its completion becomes:

```
p ↔ ¬q, ¬p                                                    1
q ↔ q                                                         2
```

which has a unique 2-valued model in which q is true and p is false. On the other hand, after adding to Π another 'meaningless' clause p p its completion becomes:

$$p \leftrightarrow p \vee (\neg q \wedge \neg p)$$
$$\neg q$$

which has a unique, yet different, 2-valued model in which q is false and p is true.

2.2.1 Three-Valued Extensions

[Fit85] showed that the inconsistency problem for Clark's semantics, as well as some other related problems, can be elegantly eliminated by considering 3-valued Herbrand models of the Clark predicate completion comp(Π), rather than 2-valued models only.

Theorem 2.2.1 (Theorem 6.1 [Fit85])**.** *Clark's completion comp(Π) of any logic program Π always has at least one 3-valued Herbrand model. Moreover, among all 3-valued models of comp(Π) there is exactly one F-least model M_Π.*

This result gave rise to Fitting's 3-valued extension of Clark's semantics.

Definition 2.2.2 (Fitting's Semantics)**.** *[Fit85] Fitting's 3-valued extension of the Clark predicate completion semantics is the semantics determined by the unique intended model M_Π or, equivalently, by the set MOD(Π) of intended models, consisting of all 3-valued Herbrand models of comp(Π).*

For example, the program 2.2.2 defined before has a unique 3-valued model in which q is false and p is undefined. Fitting also provided an elegant fixed-point characterization of 3-valued models of comp(Π).

Definition 2.2.3 (The Fitting Operator)**.** *[Fit85] Suppose that Π is a logic program. The Fitting operator $\Phi : \mathcal{I} \to \mathcal{I}$ on the set $\mathcal{I}$ of all 3-valued interpretations of comp(Π) is defined as follows. If $I \in \mathcal{I}$ is an interpretation of comp(Π) and A is a ground atom then $\Phi(I)$ is an interpretation given by*[4]*:*

(i) $\Phi(I)(A) = 1$ *if there is a clause* $A \leftarrow L_1, ..., L_n$ *in* Π *such that* $\hat{I}(L_i) = 1$*, for all* $i \leq n$*;*

(ii) $\Phi(I)(A) = 0$ *if for every clause* $A \leftarrow L_1, ..., L_n$ *in* Π *there is an* $i \leq n$ *such that* $\hat{I}(L_i) = 0$*;*

(iii) $\Phi(I)(A) = \frac{1}{2}$*, otherwise.*

Theorem 2.2.2. *[Fit85] An interpretation I of comp(Π) is a model of comp(Π) if and only if it is a fixed point of the operator Φ. In particular, M_Π is the F-least fixed point of Φ.*

Moreover, the model M_Π can be obtained by iterating the operator Φ, namely, the sequence $\Phi^{\uparrow\alpha}$ of iterations[5] *of Φ is monotonically increasing and it has a fixed point*

$$\Phi^{\uparrow\lambda} = M_\Pi.$$

Kunen [Kun87] showed that the set of formulae implied by Fitting's semantics is not recursively enumerable and he proposed the following modification of Fitting's approach.

Definition 2.2.4 (Kunen's Semantics)**.** *[Kun87] Kunen's 3-valued extension of the Clark predicate completion semantics is the semantics determined by the set MOD(Π) of intended models, consisting of all 3-valued (Herbrand or not) models of comp(Π).*

Kunen showed that his semantics is recursively enumerable and closely related to the Fitting operator Φ.

Theorem 2.2.3. *[Kun87] A closed formula V is implied by Kunen's 3-valued extension of the Clark predicate completion semantics if and only if it is satisfied in at least one finite iteration $\Phi^{\uparrow n}$ of the Fitting operator Φ, n = 0,1,2, ... Moreover, the set of formulae implied by this semantics is recursively enumerable.*

It is easy to see that Fitting's semantics is stronger than Kunen's semantics, i.e., any closed formula implied by Kunen's semantics is also implied by Fitting's semantics.

[4] According to the conventions adopted in Sec. 2.1, Π is assumed to be instantiated and interpretations are viewed as 3-valued functions.

[5] According to the convention from Sec. 2.1.1, Φ is defined on the F-ordered set of interpretations and the iteration begins from the F-smallest interpretation $I = < \emptyset, \emptyset >$.

2.2.2 Drawbacks of Clark's Completion Semantics

Unfortunately, Clark's predicate completion does not always result in a satisfactory semantics. For many programs, it leads to a semantics which appears too weak. This problem applies both to standard Clark's semantics as well as to its 3-valued extensions and it has been extensively discussed in the literature (see e.g. [She88, She84, Prz89b, GRS91]). We illustrate it on the following three examples.

Example 2.2.1 Suppose that to the program 2.2.1 defined before we add a seemingly meaningless clause:

$$\text{natural\_number(X)} \leftarrow \text{natural\_number(X)}.$$

to obtain the program 2.2.4.

Program 2.2.4

```
natural_number(0).                                              1
natural_number(succ(X)) ← natural number (X).                   2
natural_number(X) ← natural_number(X).                          3
```

It appears that the newly obtained program 2.2.4 should have the same semantics. However, Clark's completion of the new program 2.2.4 is:

$$\text{natural number(T)} \leftrightarrow$$
$$(\text{natural number(T)} \vee \text{T} = 0 \vee \exists \text{X}.\ (\text{T} = \text{succ(X)} \wedge \text{natural\_number(X)}))$$

from which it no longer follows that MickeyMouse (or anything else, for that matter) is not a natural number.

Example 2.2.2 (Van Gelder) Suppose, that we want to describe which vertices in a graph are reachable from a given vertex a. We could write the program 2.2.5.

Program 2.2.5

```
edge(a, b)                                                      1
edge(c, d)                                                      2
edge(d, c)                                                      3
reachable(a)                                                    4
reachable(X ) ← reachable(Y), edge(Y, X).                       5
```

We clearly expect vertices c and d not to be reachable. However, Clark's completion of the predicate 'reachable' gives only

$$\text{reachable(X)} \leftrightarrow (\text{X} = \text{a} \vee \exists\ \text{Y}\ (\text{reachable(Y)} \wedge \text{edge(Y, X)}))$$

from which such a conclusion again cannot be derived. Here, the difficulty is caused by the existence of symmetric clauses edge(c, d) and edge(d, c).

Example 2.2.3 Suppose the following program:

Program 2.2.6

```
bird(tweety)                                                    1
fly(X) ← bird(X), ¬abnormal(X)                                  2
abnormal(X) ← irregular(X)                                      3
irregular(X) ← abnormal(X).                                     4
```

The last two clauses merely state that irregularity is synonymous with abnormality. Based on the fact that nothing leads us to believe that tweety is abnormal, we are justified to expect that tweety flies, but Clark's completion of program 2.2.6 yields

$$\text{fly (T)} \leftrightarrow (\text{bird(T)} \vee \neg\text{abnormal(T)})$$
$$\text{abnormal(T)} \leftrightarrow \text{irregular(T)},$$

from which it does not follow that anything flies. On the other hand, without the last two clauses (or without just one of them) Clark's semantics produces correct results.

The above described behavior of Clark's completion is bound to be confusing for a thoughtful logic programmer, who may very well wonder why, for example, the addition of a seemingly harmless statement "natural_number(X) ← natural_number(X)" should change the meaning of the first program. The explanation that will most likely occur to him will be procedural in nature, namely, the fact that the above added clause may lead to a loop. But it was the idea of replacing procedural programming by declarative programming, that brought about the concept of logic programming and deductive databases in the first place, and therefore it seems that such a procedural explanation should be flatly rejected.

Some of the problems mentioned above are caused by the difficulties with the representation of transitive closures when using Clark's semantics (e.g., in the program 2.2.5). [Kun87] formally showed that Clark's semantics is not sufficiently expressive to naturally represent transitive closures. In the following sections we discuss model-theoretic approaches to declarative semantics of logic programs which attempt to avoid the drawbacks of Clark's semantics discussed above.

2.3 Least Model Semantics

The model-theoretic approach is particularly well-understood in the case of positive logic programs. In this section we assume that all interpretations are 2-valued.

Example 2.3.1 Suppose that our program Π (taken from [PP90]) consists of clauses:

Program 2.3.1

```
able mathematician(X) ← physicist(X).                           1
physicist(einstein).                                            2
businessman(iacocca).                                           3
```

This program has several different models, the largest of which is the model in which both Einstein and Iacocca are at the same time businessmen, physicists and good mathematicians. This model hardly seems to correctly describe the intended meaning of Π. Indeed, there is nothing in this program to imply that Iacocca is a physicist or that Einstein is a businessman. In fact, we are inclined to believe that the lack of such information indicates that we can assume the contrary.

The program also has the unique least model M_Π :

{physicist(einstein), businessman(iacocca), able_mathematician(einstein)},

in which only Einstein is a physicist and good mathematician and only Iacocca is a businessman. This model seems to correctly reflect the semantics of P, at the same time incorporating the classical case of the closed-world assumption [Rei77] if no reason exists for some positive statement to be true, then we are allowed to infer that it is false. It turns out that the existence of the unique least model M_Π is the property shared by all positive programs.

Theorem 2.3.1. *[vEK76b] Every positive logic program* Π *has a unique least (Herbrand) model* M_Π.

This important result led to the definition of the so called least model semantics for positive programs.

Definition 2.3.1 (Least Model Semantics)**.** *[vEK76b] By the least model semantics of a positive program* Π *we mean the semantics determined by the least Herbrand model* M_Π *of* Π.

The least Herbrand model semantics is very intuitive and it seems to properly reflect the intended meaning of positive logic programs. The motivation behind this approach is based on the idea that we should minimize positive information as much as possible, limiting it to facts explicitly implied by Π, and making everything else false. In other words, the least model semantics is based on a natural form of the closed world assumption.

The least model semantics avoids the drawbacks of the Clark predicate completion discussed in the previous section. For example, the least Herbrand model M_Π of the programs 2.2.1 and 2.2.4 given above is:

{natural_number(0), natural_number(succ(0)), natural_number(succ(succ(0))), ...}

which is exactly what we intended. Similarly, the least Herbrand model M_Π of the program 2.2.5 above is:

{edge(a, b), edge(c, d), edge(d, c), reachable(a), reachable(b)},

which is again exactly what we would expect.

Least model semantics also has a natural fixed point characterization. First we define the Van Emden-Kowalski immediate consequence operator $\Psi : \mathcal{I} \to \mathcal{I}$ on the set $\mathcal{I}$ of all interpretations of Π (ordered by $\preccurlyeq$).

Definition 2.3.2 (The Van Emden-Kowalski Operator). *[vEK76b] Suppose that* Π *is a positive logic program,* $I \in \mathfrak{I}$ *is an interpretation of* Π *and A is a ground atom. Then* $\Psi(I)$ *is an interpretation given by:*

(i) $\Psi(I)(A) = 1$ *if there is a clause* $A \leftarrow A_1, ..., A_n$ *in* Π *such that* $I(A_i) = 1$*, for all* $i \leq n$*;*

(ii) $\Psi(I)(A) = 0$ *, otherwise.*

Theorem 2.3.2. *[vEK76b] The Van Emden-Kowalski operator* Ψ *has the least fixed point, which coincides with the least model* M_Π*. Moreover, the model* M_Π *can be obtained by iterating* ω *times the operator* Ψ*, namely, the sequence* $\Psi^{\uparrow n}$*,*
$n = 0, 1, 2, ..., \omega$*, of iterations*[6]
of Ψ *is monotonically increasing and it has a fixed point* $\Psi^{\uparrow\omega} = M_\Pi$.

The least model semantics is strictly stronger than Clark's semantics:

Theorem 2.3.3. *Suppose that* Π *is a positive logic program. If a closed formula is implied by the Clark predicate completion semantics (or by one of its 3-valued extensions) then it is also implied by the least model semantics.*

The only serious, drawback of the least model semantics seems to be the fact that it is well defined for a very restrictive class of programs. Programs which are not positive, in general, do not have least models. For example, the program $p \leftarrow \neg q$ has two minimal models {p} and {q}, but it does not have the least model. Similarly, the program 2.2.6 from Example 2.2.3 does not have the least model.

2.4 Perfect Model Semantics

As we have seen above, although the least model semantics seems suitable for the class of positive programs, it is not adequate for more general programs, allowing negative premises in program clauses. The inclusion of negation in program clauses increases the expressive power of logic programs and thus is of great practical importance. At the same time, the problem of finding a suitable semantics for programs with negation becomes much more complex. In this section we discuss the perfect model semantics, which extends the least model semantics to a wider class of logic programs. Throughout most of this section by an interpretation (model) we mean a 2-valued interpretation (model).

Example 2.4.1 Suppose that we know that physicists are able mathematicians, whereas typical businessmen tend to avoid (advanced) mathematics in their work, unless they somehow happen to have a strong mathematical background. Suppose also that we know that Iacocca is a businessman and that Einstein is a physicist. We can express these facts using a logic program as follows:

[6] With respect to the standard ordering $\preccurlyeq$ of interpretations and beginning from the smallest interpretation $< \emptyset, \mathcal{H} >$. Here ω denotes the first infinite ordinal.

Program 2.4.1

```
avoids_math(X) ← businessman(X), ¬able_mathematician(X)          1
able_mathematician(X) ← physicist(X)                             2
businessman(iacocca)                                             3
physicist(einstein).                                             4
```

This program does not have a unique least model, but instead it has two minimal models (M_1 and M_2). In both of them Iacocca is the only businessman, Einstein is the only physicist and he is also an able mathematician, who uses advanced mathematics. However, in one of them, say in M_1, Iacocca avoids advanced mathematics, because he is not an able mathematician and in the other, M_2, the situation is opposite and Iacocca is an able mathematician, who uses advanced mathematics in his work.

Since any intended semantics for logic programs must include some form of the closed world assumption, and thus it must in some way minimize positive information, it is natural to consider minimal models of our program Π [Min82, BS85, McC80] as providing the desired meaning of Π. It seems clear, however, that not both minimal models capture the intended meaning of Π. By placing negated predicate able_mathematician(X) among the premises of the rule, we intended to say that businessmen, in general, avoid advanced mathematics unless they are known to be good mathematicians. Since we have no information indicating that Iacocca is a good mathematician we are inclined to infer that he does not use advanced mathematics. Therefore, only the first minimal model M_1 seems to correspond to the intended meaning of Π.

The reason for this asymmetry is easy to explain. The first clause of Π is logically (classically) equivalent to the fifth clause

```
able_mathematician(X) ∨ avoids_math(X) ← businessman(X)          5
```

and models M_1 and M_2 are therefore also minimal models of the theory Π' obtained from Π by replacing the first clause by the fifth one. However, the intended meaning of these two clauses seems to be different. The fifth clause does not assign distinct priorities to predicates (properties) able_mathematician and avoids math and thus treats them as equally plausible. As a result the semantics determined by the two minimal models M_1 and M_2 seems to be perfectly adequate to represent the intended meaning of Π'. On the other hand, the program's first clause intuitively seems to assign distinct priorities for minimization to predicates able_mathematician and avoids_math, essentially saying that the predicate able_mathematician has to be first assumed false unless there is a compelling reason to do otherwise. We can say, therefore, that the first clause assigns a higher priority for minimization (or falsification) to the predicate able_mathematician than to the predicate avoids_math.

We can easily imagine the above priorities reversed. This is for instance the case in the following clause:

$$\text{able\_mathematician(X)} \leftarrow \text{physicist(X)}, \neg\text{avoids\_math(X)}$$

which says that if X is a physicist and if we have no specific evidence showing that he avoids mathematics then we are allowed to assume that he is an able_mathematician.

Here, the predicate avoids math has a higher priority for minimization than the predicate able_mathematician, i.e., it is supposed to be first assumed false unless there is a specific reason to do otherwise.

Also observe, that if B $\leftarrow$ A is a clause, then minimizing B (i.e., making B false) immediately results in A being minimized, too. Consequently, A is always minimized before or at the same time when B is minimized. The above discussion leads us to the conclusion that the syntax of program clauses determines relative priorities for minimization among ground atoms according to the following rules:

I) Negative premises have higher priority than the heads;

II) Positive premises have priority no less than that of the heads.

To formalize conditions I and II, we assume that the program is already instantiated and we introduce the dependency graph G_Π of Π (cf. [ABW88, vG89]), whose vertices are ground atoms, i.e., elements of the Herbrand base $\mathcal{H}$. If A and B are atoms, then there is a directed edge in G_Π from B to A if and only if there is a clause in Π, whose head is A and one of whose premises is either B or $\neg$B. In the latter case the edge is called negative.

Definition 2.4.1 (Priority Relation)**.** *[ABW88] For any two ground atoms A and B in $\mathcal{H}$ we define B to have a higher priority*[7] *than A (A $<$ B) if there is a directed path in G leading from B to A and passing through at least one negative edge. We call the above defined relation $<$ the priority relation between (ground) atoms. We will write A $\leq$ B if there is a directed path from B to A.*

Analogously, we can define the predicate priority relation $<_P$ between predicate symbols, replacing in the above definition ground atoms by predicate symbols. Having defined the priority relation, we are prepared to define the notion of a perfect model. It is our goal to define a minimal model in which atoms of higher priority are minimized (or falsified) first, even at the cost of including in the model (i.e., making true in it) some atoms of lower priority. It follows, that if M is a model of Π and if a new model N is obtained from M, by adding and subtracting from M some atoms, then we will consider the new model N preferable to M if and only if the addition of any atom A is always justified by the removal of a higher priority atom B (i.e. such that A $<$ B). A model M of Π will be considered perfect, if there are no models of Π preferable to it. More formally:

Definition 2.4.2 (Perfect Models)**.** *[Prz88, Prz89b] Suppose that M and N are two distinct models of a logic program Π. We say that N is preferable to M (briefly, $N << M$), if for every atom A $\in$ N - M there is a higher priority atom B, B $>$ A, such that B $\in$ M - N . We say that a model M of Π is perfect if there are no models preferable to M. We call the relation $<<$ the preference relation between models.*

[7] There is no consensus in the literature as to whether to describe this property as having 'higher' or 'lower' priority and, accordingly, as to whether to denote it by A $<$ B or A $>$ B.

It is easy to prove

Theorem 2.4.1. *[Prz88] Every perfect model is minimal. For positive programs the concepts of a least model and a perfect model coincide.*

Example 2.4.2 Only model M_1 in Example 2.4.1 is perfect. Indeed (using obvious abbreviations):

M_2 = {physicist(e), able_mathematician(e), businessman(i), able_mathematician(i)}
M_1 = {physicist(e), able_mathematician(e), businessman(i), avoids_math(i)}

and we know that *able_mathematician* $>$ *avoids_math* and therefore $M_1 << M_2$, while not $M_2 << M_1$. Consequently, M_1 is perfect, but M_2 is not.

Unfortunately, not every logic program has a perfect model:

Example 2.4.3 The program:

```
p ← ¬ q                                                    1
q ← ¬ p                                                    2
```

has only two minimal Herbrand models $M_1 = \{p\}$ and $M_2 = \{q\}$ and since $p < q$ and $q < p$ we have $M_1 << M_2$ and $M_2 << M_1$, thus none of the models is perfect.

The cause of this peculiarity is quite clear. The concept of a perfect model is based on relative priorities between ground atoms and therefore we have to be consistent when assigning those priorities to avoid priority conflicts (cycles), which could render our semantics meaningless. This observation underlies the approaches of Apt, Blair and Walker [ABW88] and Van Gelder [vG89] who argued that when using negation we should be referring to an already defined relation, so that the definition is not circular, or, as Van Gelder puts it, we should avoid negative recursion. This idea led them to the introduction of the class of stratified logic programs (see also [CH85, Naq86]). The class of stratified logic programs has been later extended [Prz88] to the class of locally stratified programs.

Definition 2.4.3. *[ABW88, vG89, Prz88] A logic program* Π *is stratifed (resp. locally stratified) if it is possible to decompose the set S of all predicate symbols (resp. the Herbrand base* $\mathcal{H}$*) into disjoint sets* S_1, S_2, ... , S_α, ..., $\alpha < \lambda$, *called strata, so that for every clause (resp. instantiated clause):*

$$C \leftarrow A_1, ..., A_m, \neg B_1 , ..., \neg B_n$$

in P, where A's, B's and C are atoms, we have that:

i) for every i, stratum(A_i) $\leq$ *stratum(C);*

ii) for every j, stratum(B_j) $<$ *stratum(C),*

where stratum(A) $= \alpha$, *if the predicate symbol of A belongs to* S_α *(resp. if the atom A belongs to* S_α*). Any particular decomposition* $\{S_1, ..., S_\alpha, ...\}$ *satisfying the above conditions is called a stratification of* Π *(resp. local stratification of* Π*).*

In the above definition, stratification determines priority levels (strata), with lower level (stratum) denoting higher priority for minimization. For example, the program from Example 2.4.1 is stratified and one of its stratifications is

$$S_1 = \{\text{able\_mathematician}\},$$
$$S_2 = \{\text{businessman, physicist, avoids\_math}\}.$$

The difference between the definitions of stratification and local stratification is that in the first case we decompose the set S of all predicate symbols, while in the second case we decompose the Herbrand base $\mathcal{H}$. Since every program can effectively refer only to a finite set of predicate symbols, stratifications can be always assumed to be finite. On the other hand, if the program uses function symbols then its Herbrand universe is infinite and its local stratifications can, in general, be infinite. The following fact is obvious:

Proposition 2.4.1. *Every stratified program is locally stratified.*

The next proposition characterizes (local) stratifiability.

Proposition 2.4.2. *[ABW88, Prz88] A logic program* Π *is stratified if and only if its predicate priority relation* $<_\Pi$ *is a partial order*[8]*. A logic program* Π *is locally stratified if and only if its priority relation* $<$ *is a partial order and if every increasing sequence of ground atoms under* $<$ *is finite*[9]*.*

All programs described in sections 2.2, 2.3 and 2.4, with the exception of Example 2.4.3, are stratified. The program in Example 2.4.3, is not even locally stratified. We now present an example of a locally stratified program which is not stratified.

Example 2.4.4 The following program defines even numbers:

```
even(0)                                    1
even(s(X)) ← even(X).                      2
```

Here s(X) is meant to represent the successor function on the set of natural numbers. This program is not stratified because the predicate even is involved in negative recursion with itself, i.e., even $<_\Pi$ even. However, Π is locally stratified, because the priority ordering $<$ between ground atoms is easily seen to be a partial order and every increasing sequence of ground atoms is of the form:

$$even(s(s(s(...)))) < even(s(s(...))) < even(s(...)) < ... < even(s(0)) < even(0)$$

and therefore it must be finite.

The following basic result shows that every locally stratified program has the least model M_Π with respect to the preference relation $<<$.

Theorem 2.4.2. *[Prz88] Every locally stratified program* Π *has a unique perfect model* M_Π*. Moreover,* M_Π *is preferred to any other model* M *of* Π*, i.e.,* $M_\Pi << M$*, for any other model* M*.*

[8] By a partial order we mean an irreflexive and transitive relation.

[9] The last condition is only essential when the Herbrand base is infinite.

For stratified programs, models M_Π have been first introduced under the name of 'natural' models in [ABW88, vG89] and defined in terms of iterated fixed points and iterated least models. In general, a (locally) stratified program may have many stratifications, however, the notion of a perfect model is defined entirely in terms of the priority relation $<$ and thus it does not depend on a particular stratification. Now we can define the perfect model semantics of locally stratified logic programs.

Definition 2.4.4 (Perfect Model Semantics)**.** *[ABW88, vG89, Prz88] Let Π be a locally stratified*[10] *logic program. By the perfect model semantics of Π we mean the semantics determined by the unique perfect model M_Π of Π. It follows immediately from Theorem 2.4.1 that for positive logic programs the perfect model semantics is in fact equivalent to the least model semantics and thus the perfect model semantics extends the least model semantics. The following result, slightly generalizing [ABW88], shows that the perfect model semantics is strictly stronger than the semantics defined by Clark's completion.*

Theorem 2.4.3. *(cf. [ABW88]) If Π is a locally stratified logic program, then Clark's completion comp(Π) is consistent and if a closed formula is implied by the Clark predicate completion semantics (or by one of its 3-valued extensions) then it is also implied by the perfect model semantics.*

The perfect model semantics eliminates various unituitive features of Clark's semantics discussed before. For example, the unique perfect model of the program 2.2.6 discussed in Example 2.2.3 consists of:

$$\{\ \text{bird(tweety)},\ \text{fly(tweety)}\ \},$$

leading to the expected intended semantics.

2.4.1 Perfect Models As Iterated Fixed Points and Iterated Least Models

Least models of positive programs have been characterized as fixed points of the Van Emden-Kowalski operator. It turns out that perfect models of locally stratified programs can be also characterized as iterated least fixed points and as iterated least models of the program. In the remainder of this section we consider both 2-valued and 3-valued interpretations.

First we need a generalization of the Van Emden-Kowalski operator Ψ defined in Section 2.3. For any interpretation J we define a corresponding operator Ψ_J as follows:

Definition 2.4.5 (The Generalized Van Emden-Kowalski Operator)**.** *Suppose that Π is any logic program, I, J $\in \mathfrak{I}$ are interpretations and A is a ground atom. Then $\Psi_J(I)$ is a (2-valued) interpretation given by:*

(i) $\Psi_J(I)(A) = 1$ if there is a clause $A \leftarrow L_1, ..., L_n$ in Π such that, for all $i \leq n$, either $\hat{J}(L_i) = 1$ or L_i is an atom and $I(L_i) = 1$;

[10] Perfect model semantics can be defined for a significantly larger class of programs, but for the sake of compatibility with its extensions discussed in the following sections, we limit it to the class of locally stratified programs.

(ii) $\Psi_J(I)(A) = 0$, *otherwise.*

Intuitively, J represents facts currently known to be true or false and Ψ_J(I) contains all atomic facts whose truth can be derived in one step from the program Π assuming that all facts in J hold and assuming that all positive facts in I are true. The Van Emden-Kowalski operator Ψ coincides with Ψ_J, where $J =< \emptyset, \emptyset >$. Observe, that operators Ψ_J are asymmetric in the sense that they do not treat negative and positive information symmetrically. In Section 2.5 devoted to the well-founded semantics we will introduce analogous, but completely symmetric operators.

Proposition 2.4.3. *[ABW88] For every interpretation J , the operator* Ψ_J *is monotone and it has the least fixed point given by* $\Psi_J^{\uparrow\omega}$ *(recall that* ω *stands for the first infinite ordinal).*

Intuitively, the least fixed point $\Psi_J^{\uparrow\omega}$ contains all positive (atomic) facts which can be derived from Π knowing J. Now we give an iterated fixed point characterization of perfect models. Let $\{S_1, S_2, \ldots, S_\alpha, \ldots\}$, $\alpha < \lambda$, be a local stratification of a program Π, i.e., a decomposition of the Herbrand base $\mathcal{H}$. For every $\beta \leq \lambda$ let

$$\mathcal{H}_\beta = \bigcup_{\alpha \leq \beta} S_\alpha.$$

Clearly,

$$\mathcal{H} = \mathcal{H}_\lambda.$$

Since the result of applying an operator Ψ_J to an arbitrary interpretation I is always a 2-valued interpretation Ψ_J(I), we can identify interpretations Ψ_J(I) with subsets of the Herbrand base. We construct the following (transfinite) sequence $\{ I_\alpha : \alpha \leq \lambda \}$ of interpretations:

$$I_0 =< \emptyset, \emptyset >$$
$$I_{\alpha+1} =< \Psi^\omega_{I_\alpha}, \mathcal{H}_{\alpha+1} - \Psi^\omega_{I_\alpha} >$$
$$I_\delta = \sum_F \{ I_\alpha : \alpha \leq \delta \},$$

for limit δ. At any given step α, the next iteration $I_{\alpha+1}$ is obtained by:

- Taking the least fixed point $\Psi^\omega_{I_\alpha}$ of the operator Ψ_{I_α} as the set of positive facts. This is justified by the fact that the least fixed point $\Psi^\omega_{I_\alpha}$ contains those positive facts whose truth can be deduced from Π assuming I_α.
- Taking the complement of $\Psi^\omega_{I_\alpha}$ in $\mathcal{H}_{\alpha+1}$ as the set of false facts. This is justified by the fact that the program is locally stratified and thus atoms from $\mathcal{H}_{\alpha+1}$, whose truth cannot be deduced at the level $\alpha + 1$ can be assumed to be false.

One can show that the sequence $\{ I_\alpha \}$ is F-increasing and, clearly, the last interpretation in this sequence is I_λ. Observe again, that the above definition of iterations I_α does not treat negative and positive information symmetrically. In Section 2.5 devoted to the well-founded semantics we will give an analogous, but symmetric definition.

The following two theorems generalize results obtained in [ABW88, vG89] from the class of stratified programs to the class of locally stratified programs. The approach presented here is slightly different from those given in [ABW88, vG89]

Theorem 2.4.4. *(cf. [ABW88, vG89]) The unique perfect model M_Π of a locally stratified program coincides with I_λ. Moreover, M_Π is itself the least fixed point of the operator Ψ_{M_Π}.*

Thus perfect models of locally stratified programs can be viewed as iterated least fixed points of operators Ψ_J. Perfect models can be also described as iterated least models of the program.

Let us first denote by Π_α the set of all (instantiated) clauses of Π whose heads belong to $\mathcal{H}_\alpha$. Clearly, $\Pi_\lambda = \Pi$. It is easy to see that if in the above definition of the sequence I_α we replace the definition of $I_{\alpha+1}$ by:

$$I_{\alpha+1} = < \mathcal{H}_{\alpha+1} \cap \Psi^\omega_{I_\alpha}, \mathcal{H}_{\alpha+1} - \Psi^\omega_{I_\alpha} >$$

i.e., if we restrict true atoms to the elements of $\mathcal{H}_{\alpha+1}$ then we will still have $M_\Pi = I_\lambda$. For the so modified sequence I_α the following result holds.

Theorem 2.4.5. *(cf. [ABW88, vG89]) For every $\alpha \leq \lambda$, I_α is the least model of the program Π_α, which extends all models I_β of programs Π_β, for $\beta < \alpha$ (i.e., such that $I_\beta \preccurlyeq_F I_\alpha$).*

Thus M_Π can be viewed as an iterated least model of Π.

2.4.2 Extensions of the Perfect Model Semantics

The class of perfect models of locally stratified logic programs has many natural and desirable properties. However, the fact that it is restricted to the class of locally stratified programs is a significant drawback. Several researchers pointed out that there exist interesting and useful logic programs with natural intended semantics, which are not locally stratified [GL88, GRS91]

Example 2.4.5 [GL88] Consider the program Π given by:

```
p(1, 2) ←                                    1
q(X) ← p(X, Y), ¬q(Y).                       2
```

After instantiating, Π takes the form:

```
p(1, 2) ←                                    6
q(1) ← p(1, 2), ¬q (2)                       7
q(1) ← p(1, 1), ¬q (1)                       8
q(2) ← p(2, 2), ¬q (2)                       9
q(2) ← p(2, 1), ¬q (1).                      10
```

This program is not locally stratified. Indeed, the priority relation $<$ between atoms is not a partial order, because $q(1) < q(2)$ and $q(2) < q(1)$. On the other hand, it seems clear that the intended semantics of Π is well-defined and is characterized by the 2-valued model $M = \{$ p(1, 2), q(1) $\}$ of Π. The same results would be produced by Prolog, which further confirms our intuition.

The cause of this peculiarity is fairly clear. Program Π appears to be semantically equivalent to a locally stratified program Π* consisting only of clauses (6) and (7).

The remaining clauses seem to be entirely irrelevant, because atoms p(1, 1), p(2, 1) and p(2, 2) can be assumed false in Π. At the same time, they are the ones that destroy local stratifiability of Π.

Three different extensions of the perfect model semantics have been proposed: the stable model semantics [GL88] (equivalent to the default model semantics [BF91]) the weakly perfect model semantics [PP88] and the well-founded semantics [GRS91]. While the first two semantics are 2-valued and are defined only for restricted classes of programs, the well-founded semantics is 3-valued and is defined for all logic programs. Although the three semantics approach the problem from three different angles (see [PP90]) it appears, that the well-founded semantics is the most adequate extension of the perfect model semantics. In the next section we discuss the Well Founded Semantics.

2.5 Well Founded Model Semantics

The well-founded semantics has been introduced in [GRS91] and it seems to be the most adequate extension of the perfect model semantics to the class of all logic programs, avoiding various drawbacks of the other proposed approaches. Well-founded semantics also has been shown to be equivalent to suitable forms of 3-valued formalizations of all four major non-monotonic formalisms [Prz89c]

One of the important features of well-founded models, and a strong indication of their naturality, is the fact that they can be described in many different, but equivalent, ways (see [GRS91, Prz89a, Prz89c, Gel89, Bry89]). Here we use the (iterated) least fixed point approach proposed in [Prz89a], which seems to be a natural extension of least fixed point definitions of least models and perfect models and is also closely related to Fitting's extension of Clark's semantics. As opposed to the original definition proposed in [GRS91], the iterated fixed point definition given here is constructive.

First, for any interpretation $\mathcal{J}$ of a program Π, we introduce the operator $\Theta_J : \mathcal{I} \rightarrow \mathcal{I}$ on the set I of all 3-valued interpretations of Π , ordered by the standard ordering $\preccurlyeq$. The operator can be viewed as a cross between the Fitting operator Φ (Sec. 2.2) and Generalized Van Emden-Kowalski operators Ψ_J (see Sec. 2.4).

Definition 2.5.1. *[Prz89a] Suppose that Π is a logic program and J is its interpretation. The operator $\Theta_J : \mathcal{I} \rightarrow \mathcal{I}$ on the set I of all 3-valued interpretations of Π is defined as follows. If $I \in \mathcal{I}$ is an interpretation of Π and A is a ground atom then $\Theta_J(I)$ is an interpretation given by:*

(i) *$\Theta_J(I)(A) = 1$ if there is a clause $A \leftarrow L_1, ..., L_n$ in Π such that, for all $i \leq n$, either $\hat{J}(L_i) = 1$ or L_i is positive and $I(L_i) = 1$;*

(ii) *$\Theta_J(I)(A) = 0$ if for every clause $A \leftarrow L_1, ..., L_n$ in Π there is an $i \leq n$ such that either $J(L_i) = 0$ or L_i is positive and $I(L_i) = 0$;*

(iii) *$\Theta_J(I)(A) = \frac{1}{2}$, otherwise.*

Intuitively, the interpretation J represents facts currently known to be true or false. The true facts in $\Theta_J(I)$ consist of those atoms which can be derived in one step from the program Π assuming that all facts in J hold and that all positive facts in I are true. The false facts in $\Theta_J(I)$ consist of those atoms whose falsity can be deduced in one step (using the closed world assumption) from the program Π assuming that all facts in J hold and that all negative facts in I are true. Observe, that, as opposed to Van Emden-Kowalski operators Ψ_J, the operators Θ_J are completely symmetric in the sense that they treat negative and positive information symmetrically (dually).

Theorem 2.5.1. *[Prz89a] For every interpretation J, the operator Θ_J is monotone and it has a unique least fixed point*[11] *given by $\Theta_J^{\uparrow\omega}$.*

We will denote this least fixed point of Θ_J by $\Omega(J)$, i.e.

$$\Omega(J) = \Theta_J^{\uparrow\omega}$$

Clearly, Ω constitutes an operator on the set of all interpretations of Π.

Observe that, although the operators Θ_J resemble the Fitting operator Φ, they do not coincide with it. Moreover, the above Theorem is very different from theorem 2.2.2. This is a consequence of the fact that the operators Θ_J are defined on the set of all interpretations ordered by the standard ordering $\preccurlyeq$ and not by the F-ordering $\preccurlyeq_F$ and thus the iterations begin from the smallest interpretation $I_0 =< \emptyset, \mathcal{H} >$ and not from the F-smallest interpretation $< \emptyset, \emptyset >$. As a result, as opposed to Φ, least points of operators Θ_J can be always obtained after only ω steps, where ω is the first infinite ordinal.

Intuitively, the least fixed point $\Omega(J) = \Theta_J^{\uparrow\omega}$ of Θ_J contains all facts, whose truth or falsity can be deduced (using the closed world assumption) from Π knowing J. The operator Ω turns out to have a unique F-least fixed point.

Theorem 2.5.2. *[Prz89a] The operator always has a unique F-least fixed point M_Π, i.e. the F-least interpretation M_Π such that*

$$\Omega(M_\Pi) = M_\Pi.$$

Moreover, all fixed points of Ω are minimal models of Π.

As we will see below, the F-least fixed point of Ω can be simply obtained as a suitable iteration $\Omega^{\uparrow\lambda}$ of Ω. First, we give a definition of well-founded models.

Definition 2.5.2. *[Prz89a] We call the unique F-least fixed point M_Π of Ω the well founded model of Π.*

Since the well-founded model of Π is defined as the F-least fixed point of the operator Ω, which is itself defined by means of least fixed points of Θ, well founded models can be viewed as iterated least fixed points of the operator Θ. Although our definition of well-founded models is different from the original definition given in [GRS91], the two notions are equivalent.

[11] Recall that ω stands for the first infinite ordinal and that the iteration begins from the smallest interpretation $I_0 =< \emptyset, \mathcal{H} >$.

Theorem 2.5.3. *[Prz89a] Well founded models introduced above coincide with well-founded models originally defined in [GRS91].*

Now we can introduce the well-founded semantics of logic programs.

Definition 2.5.3 (Well-Founded Semantics)**.** *[GRS91], The well founded semantics of a logic program is determined by the unique well-founded model* M_Π.

In order to obtain a constructive definition of the well-founded model M_Π of a given program Π , we define the following (transfinite) sequence $\{\ I_\alpha\ \}$ of interpretations of Π:

$$\begin{gathered} I_0 =< \emptyset, \emptyset > \\ I_{\alpha+1} = \Omega(I_\alpha) = \Theta_{I_\alpha}^{\uparrow\omega} \\ I_\delta = \sum_F \{I_\alpha : \alpha < \delta\}, \end{gathered}$$

for limit δ. Clearly, for any α, I_α coincides with $\Omega^{\uparrow\alpha}$, where the F-ordering of interpretations is used to generate consecutive iterations, i.e., we have:

$$I_\alpha = \Omega^{\uparrow\alpha}.$$

At any given step α, the next iteration $I_{\alpha+1}$ is obtained as the least fixed point $\Omega(I_\alpha) = \Theta_{I_\alpha}^\omega$ of the operator Θ_{I_α}. This is justified by the fact that, as we observed before, the least fixed point $\Omega(I_\alpha) = \Theta_{I_\alpha}^\omega$ contains all facts, whose truth or falsity can be deduced from Π knowing I_α.

One can show that the sequence $\{\ I_\alpha\ \}$ is well-defined and F-increasing and therefore, since all interpretations are countable, there must exist the smallest λ, such that I_λ is a fixed point , i.e., such that:

$$I_{\lambda+1} = \Omega(I_\lambda) = I_\lambda$$

i.e., I_λ is a fixed point of the the operator Ω. We call $\lambda = \lambda(\Pi)$ the depth of the program Π. It turns out that I_λ is in fact the F-least fixed point of Ω and thus it coincides with the well-founded model M_Π of Π.

Theorem 2.5.4. *[Prz89a] The interpretation* $I_\lambda = \Omega^{\uparrow\lambda}$ *is the F-least fixed point of the operator* Ω *and thus it coincides with the well-founded model* M_Π *of* Π*:*

$$M_\Pi = I_\lambda = \Omega^{\uparrow\lambda}$$

Observe, that the above description of well-founded models is very similar to the iterated fixed point definition of perfect models given in Sec. 2.4, but it treats negative and positive information completely symmetrically (dually) and does not require the advance notion of (local) stratification. In particular, the well-founded model semantics extends the perfect model semantics.

Corollary 2.5.1. *[GRS91] The well-founded model of a locally stratified program coincides with its perfect model.*

In [Prz89a] the iterated fixed point definition of well founded models is used to introduce the so called dynamic stratification of an arbitrary logic program Π, with properties analogous to local stratification. Using dynamic stratification, [Prz89a] showed that the well-founded model M_{Π} can also be viewed as an iterated least model of a program and that well-founded models can be defined by means of a suitable preference relation between atoms, in a manner analogous to the definition of perfect models.

CHAPTER 3

ON IMPLEMENTING WELL FOUNDED SEMANTICS

The central component of existing logic programming systems is a refutation procedure, which is based on the resolution rule created by Robinson [Rob65]. The first such refutation procedure, called SLD-resolution, was introduced by Kowalski [Kow74, VEK76a], and further formalized by Apt and Van Emden [AvE82]. SLD-resolution is only suitable for positive logic programs, i.e. programs without negation. Clark [Cla78] extended SLD-resolution to SLDNF-resolution by introducing the negation as finite failure rule, which is used to infer negative information. SLDNF-resolution is suitable for general logic programs, by which a ground negative literal $\neg$A succeeds if A finitely fails, and fails if A succeeds.

As an operational/procedural semantics of logic programs, SLDNF-resolution has many advantages, among the most important of which is its linearity of derivations. Let $G_0 \Rightarrow_{C_1,\Theta_1} G_1 \Rightarrow \ldots \Rightarrow_{C_i,\Theta_i} G_i$ be a derivation with G_0 the top goal and G_i the latest generated goal. A resolution is said to be linear for query evaluation if when applying the most widely used depth-first search rule, it makes the next derivation step either by expanding G_i using a program clause (or a tabled answer), which yields $G_i \Rightarrow_{C_{i+1},\Theta_{i+1}} G_{i+1}$, or by expanding G_{i-1} via backtracking[1]. It is with such linearity that SLDNF-resolution can be realized easily and efficiently using a simple stack-based memory structure [War83, Zho94]. This has been sufficiently demonstrated by Prolog, the first and yet the most popular logic programming language which implements SLDNF-resolution.

However, SLDNF-resolution suffers from two serious problems. One is that the declarative semantics it relies on, i.e. the completion of programs [Cla78], incurs some anomalies (see [Llo87, She88] for a detailed discussion); and the other is that it may generate infinite loops and a large amount of redundant sub-derivations [BAK91, SD94, Vie89]. This problems are introduced in detail in section 3.1.

The first problem with SLDNF-resolution has been perfectly settled by the definition of the well-founded semantics [GRS91]. Three representative methods were then proposed for topdown evaluation of such a new semantics: Global SLS-resolution

[1]The concept of "linear" here is different from the one used for SL-resolution [KK71].

[Prz89a, Ros92], SLG-resolution [CSW95, CW96] and SLT-resolution [SyYhY02].

Global SLS-resolution is a direct extension of SLDNF-resolution. It overcomes the semantic anomalies of SLDNF-resolution by treating infinite derivations as failed and infinite recursions through negation as undefined. Like SLDNF-resolution, it is linear for query evaluation. However, it inherits from SLDNF-resolution the problem of infinite loops and redundant computations. Therefore, as the authors themselves pointed out, Global SLS-resolution can be considered as a theoretical construct [Prz89a] and is not effective in general [Ros92].

SLG-resolution (similarly, Tabulated SLS-resolution [BD98]) is a tabling mechanism for topdown evaluation under the well-founded semantics. The main idea of tabling is to store intermediate results of relevant subgoals and then use them to solve variants of the subgoals whenever needed. With tabling no variant subgoals will be recomputed by applying the same set of program clauses, so infinite loops can be avoided and redundant computations be substantially reduced [BD98, CW96, TS86, Vie89, War92]. Like all other existing tabling mechanisms, SLG-resolution adopts the solution-lookup mode. That is, all nodes in a search tree/forest are partitioned into two subsets: solution nodes and lookup nodes. Solution nodes produce child nodes only using program clauses, whereas lookup nodes produce child nodes only using answers in the tables. As an illustration, consider the derivation $p(X) \Rightarrow_{C_{p_1},\Theta_1} q(X) \Rightarrow_{C_{q_1},\Theta_2} p(Y)$. Assume that so far no answers of p(X) have been derived (i.e., currently the table for p(X) is empty). Since p(Y) is a variant of p(X) and thus a lookup node, the next derivation step is to expand p(X) against a program clause, instead of expanding the latest generated goal p(Y). Apparently, such kind of resolutions is not linear for query evaluation. As a result, SLG-resolution cannot be implemented using a simple, efficient stack-based memory structure

SLT-resolution was the next development step, offering the user a linear tabling method for top-down evaluation of the WFS of general logic programs. As SLG, it resolves infinite loops and redundant computations, but it does not sacrifice the linearity of SLDNF-resolution (like Global SLS-resolution). It is based on SLT-trees, which are basically SLDNF-trees with an enhancement of some loop handling mechanisms. Consider again the derivation $p(X) \Rightarrow_{C_{p_1},\Theta_1} q(X) \Rightarrow_{C_{q_1},\Theta_2} p(Y)$. Note that the derivation has gone into a loop since the proof of p(X) needs the proof of p(Y), a variant of p(X). By SLDNF- or Global SLS-resolution, p(Y) will be expanded using the same set of program clauses as p(X). Obviously, this will lead to an infinite loop of the form $p(X) \Rightarrow_{C_{p_1}} ...p(Y) \Rightarrow_{C_{p_1}} ...p(Z) \Rightarrow_{C_{p_1}} ...$ In contrast, SLT-resolution will break the loop by disallowing p(Y) to use the clause C_{p_1} that has been used by p(X). As a result, SLT-trees are guaranteed to be finite for programs with the bounded-term-size property.

In contrast to SLG, SLT-resolution is linear for query evaluation. Unlike SLG-resolution and all other existing top-down tabling methods, SLT-resolution does not distinguish between solution and lookup nodes. All nodes will be expanded by applying existing answers in tables, followed by program clauses. For instance, in the above example derivation, since currently there is no tabled answer available to p(Y), p(Y) will be expanded using some program clauses. If no program clauses are available to p(Y), SLT-resolution would move back to q(X) (assume using a depth-first control

strategy). This shows that SLT-resolution is linear for query evaluation. When SLT-resolution moves back to p(X), all program clauses that have been used by p(Y) will no longer be used by p(X). This avoids redundant computations.

In spite of the improvements of SLT over SLG and SLS, all these procedures are restricted to programs with the bounded-term-size property and non-floundering queries. This last problem is what we solve by combining the management of positive and negative literals in a symmetrical way from Constructive Intensional Negation [MMNMH08] and the Derivation Procedure for Extended Stable Models [PAP+91], which allows us to determine if a query belongs or not to the well founded model of the program, obtaining well founded semantics.

In the following sections we introduce Negation as Failure and its problems (Sec. 3.1), the mechanisms used in Constructive Intensional Negation (Sec. 3.2), the Derivation Procedure for Extended Stable Models (Sec. 3.3) and the general picture of our implementation (sections 3.4 and 3.5).

3.1 Negation as Failure

Negation as Failure (NAF) is the dominant mechanism for processing negative literals in logic programming. NAF, as pointed out by [BMPT90], is a meta-inference-rule allowing one to prove the negation of a ground goal, when the proof of the corresponding positive rule fails. The limitation of ground goals is a real problem, as can be seen in the following examples.

Remark. We use for negation the predicate **naf** instead of **not** to denote that we are using Negation as Failure.

Example 3.1.1 (NAF works for ground goals) In the following program, fly depends on naf(penguin) and, as we can not prove penguin (the program has only one rule), then fly is true.

Program 3.1.1

```
fly ← naf(penguin)
```

Evaluation of queries fly and $naf(fly)$ in program 3.1.1

```
?- fly.                          ?- naf(fly).
   true                             fail
```

Example 3.1.2 (NAF does not work for non-ground goals) In program 3.1.2, fly(X) depends on naf(penguin(X)). We have added Tweety and Donald. Tweety is a penguin and Donald flies because is a duck. In this way we expect a query $?- \ fly(Y)$ to get an answer $Y = donald$ and a query $?- \ naf(fly(Y))$ to get an answer $Y = tweety$. While the first query works, the second one does not.

Program 3.1.2

```
fly(X) ← ¬penguin(X)                                    1
penguin(tweety).                                        2
fly(donald).                                            3
```

Evaluation of queries $fly(Y)$ and $naf(fly(Y))$ in program 3.1.2

```
?- fly(Y).                          ?- ¬fly(Y).
   Y = donald?                         no
   no
```

The reason for this strange behavior is in the implementation of NAF:

```
naf(X) ← X, !, fail.                                    1
naf(X).                                                 2
```

While for the example 3.1.1 this works perfectly because the interpreter is not able to find the proposition fly, in the example 3.1.1 it tries to find $fly(Y)$, which is found for $Y = donald$. As it is found, the interpreter executes the first rule of the predicate naf, and fails to find a suitable value for Y making $naf(fly(Y))$ true.

Logically speaking, instead of getting the answers for the first rule below we are getting the answers for second rule.

```
∃ Y. ¬fly(Y). [Expected behavior]                        1
∀ Y. ¬fly(Y). [NAF behavior]                             2
```

As can be seen in the example, NAF is a mechanism with a procedural semantics that are far from the expected declarative semantics when programs are non-ground.

Besides, NAF does not take care of loops. The following example illustrate this problem.

Example 3.1.3 The following program naturally encodes the knowledge that we sleep if we do not work and we work if we do not sleep. As works depends on sleep and sleep depends on work, the evaluation never stops or stops with an error message.

Program 3.1.3

```
work ← naf(sleep).                                      1
sleep ← naf(work).                                      2
```

Evaluation of queries $sleep$, $work$, $naf(sleep)$ and $naf(work)$ in program 3.1.3

```
?- sleep.                           ?- naf(sleep).
   ...                                 ...
   [OutOfMemory exception]             [OutOfMemory exception]

?- work.                            ?- naf(work).
   ...                                 ...
   [OutOfMemory exception]             [OutOfMemory exception]
```

One may argue that this problem exists in Prolog without making use of negation, which is true. The fact is that, as explained in chapter 1, we can define our knowledge about these propositions as incomplete or perhaps even confusing, but in this cases the implementation should take this into account and never abort the computation. As has been introduced in chapter 2, this problem was solved by developing new semantics for logic programs. The combination of the techniques revised in sections 3.2 and 3.3, presented in section 3.4, goes in this line of research.

3.2 Constructive Intensional Negation

Here we introduce the technique developed in [MMNMH08] to obtain a variant of Clark's Predicate Completion Semantics (see Sec. 2.2) that the authors suggest to be seen as a CLP version of Kunen's Semantics. Its name is Constructive Intensional Negation (CIN).

CIN is the first working implementation of the procedure developed by Barbuti *et al*. [BMPT87, BMPT90], Intensional Negation (IN). IN uses a transformational approach that has some problems when transforming certain classes of programs, so the authors of CIN slightly modified it to make it work. We will introduce Intensional Negation and its problems to get to Constructive Intensional Negation.

In *intensional negation* [BMPT87, BMPT90] a program transformation technique is used to add new predicates to the program in order to express the negative information. Informally, the *complement* of head terms of the positive clauses are computed and they are used later as the head of the negated predicate. We denote the negated predicate of p as $intneg\_p$.

Remark. In the following example, Ex. 3.2.1, from the fact that the Peano numbers 0 and $s(s(X))$ are even (if X is even again) it is inferred that the Peano numbers $s(0)$ and $s(s(Y))$ are not even (if Y is not even again). As we will see in Sec. 4.2, this inference is done from the human knowledge that $s(0)$ and $s(s(Y))$ are not even, but it is impossible (without human intervention) for an algorithm to determine that.
This example only illustrates the ideal computation of the negative clauses. The real result is presented in Ex. 3.2.2, where the term $forall([Y], X \neq s(s(Y)))$ represents the formula $\forall Y.\ X \neq s(s(Y))$. This formula is obtained from the negation of the formula $\exists X.\ Y = s(s(X))$, that represents the head of the second clause of even, $even(s(s(X)))$. The process is explained in Def. 3.2.1.

Program 3.2.1 Predicate even for Peano numbers, from [MMNMH08]

```
even(0) ← .                                          1
even(s(s(X))) ← even (X).                            2
                                                     3
intneg_even(s(0)) ← .                                4
intneg_even(s(s(Y))) ← intneg_even(Y).               5
```

Program 3.2.2 Predicate even for Peano numbers, from [MMNMH08]

```
even(0) ← .                                                    1
even(s(s(X))) ← even (X).                                      2
                                                               3
intneg_even(X) ← X ≠ 0, forall([Y], X ≠ s(s(Y)) ).             4
intneg_even(s(s(Y))) ← intneg_even(Y).                         5
```

There is one problem with this transformation technique: in the presence of new logical variables on the right-hand side (rhs) of a clause[2], the new program needs to handle some kind of universal quantification construct (The $\forall$ symbol in Ex. 3.2.2). It is solved by using a method semantically similar to the unfolding procedure in Sato and Motoyoshi's work [ST84, Sat89], but not requiring the management of syntactic structures. The method deals with a recursive implementation of universal quantification based on an expansion theorem.

The theoretical part is justified with the definition provided for the Intensional Negation of a program, based on the extension of the signature used to build the program:

Given a signature $\Sigma = \langle FS_\Sigma, PS_\Sigma \rangle$, let $PS'_\Sigma \supset PS_\Sigma$ be such that for every $p \in PS_\Sigma$ there exists a symbol $neg(p) \in PS'_\Sigma \setminus PS_\Sigma$. Let $\Sigma' = \langle FS_\Sigma, PS'_\Sigma \rangle$.

Definition 3.2.1 (Intensional Negation of a Program)**.** *Given a program Π_Σ, its* intensional negation *is a program $\Pi'_{\Sigma'}$ such that for every p in PS_Σ the following properties hold:*

P.1) Conservativity*:* $\forall \overline{x}.\, [Comp(\Pi) \models_3 p(\overline{x}) \iff Comp(\Pi') \models_3 p(\overline{x})]$

P.2) Negation*:* $\forall \overline{x}.\, [Comp(\Pi) \models_3 p(\overline{x}) \iff Comp(\Pi') \models_3 \neg(neg(p)(\overline{x}))]$

From the definition provided they define a method able to generate the negated counterpart of a set of clauses if they are non-overlapping clauses (see def. 3.2.2), and a method to translate overlapping rules into non-overlapping ones.

Definition 3.2.2 (Nonoverlapping predicate definition)**.** *A pair of constraints c_1 and c_2 is said to be* incompatible *iff their conjunction $c_1 \wedge c_2$ is unsatisfiable. A set of constraints $\{ c_i \}$ is* exhaustive *iff $\bigvee_i c_i = \mathbf{t}$. A predicate definition*

$$def_\Pi(p) \;=\; \{p(\overline{x}) \;\leftarrow\; B_i(\overline{y}_i \cdot \overline{z}_i) \,[\!]\, c_i(\overline{x} \cdot \overline{y}_i) \mid i \in 1 \ldots m\}$$

is nonoverlapping *iff $\forall i, j \in 1 \ldots m$ the constraints $\exists \overline{y}_i.\, c_i(\overline{x} \cdot \overline{y}_i)$ and $\exists \overline{y}_j.\, c_j(\overline{x} \cdot \overline{y}_j)$ are incompatible (so, $i \neq j$) and $def_\Pi(p)$ is exhaustive if the set of constraints $\{\exists \overline{y}_i.\, c_i(\overline{x} \cdot \overline{y}_i)\}$ is exhaustive.*

A nonoverlapping set of clauses can be made into a nonoverlapping and exhaustive set by adding an extra "default" case:

[2]Programs in which all variables appearing in the body of rules also appear in the corresponding head are called covered by some authors (see [APS04]). Here by programs with new logical variables in the rhs of a clause we refer to non-covered programs.

Lemma 3.2.1. *Let p be such that its definition*

$$def_{\Pi}(p) = \{p(\overline{x}) \ \leftarrow \ B_i(\overline{y}_i \ \cdot \ \overline{z}_i) [\!] \ c_i(\overline{x} \ \cdot \ \overline{y}_i) \mid i \ \in \ 1 \ldots m\}$$

is nonoverlapping. Then its completed definition is logically equivalent to

$$\begin{aligned} cdef_{\Pi}(p) \equiv \forall \overline{x}.[p(\overline{x}) \iff & \ \exists \overline{y}_1. \ c_1(\overline{x} \cdot \overline{y}_1) \wedge \exists \overline{z}_1.B_1(\overline{y}_1 \cdot \overline{z}_1) \parallel \\ & \vdots \\ & \exists \overline{y}_m. \ c_m(\overline{x} \cdot \overline{y}_m) \wedge \exists \overline{z}_m.B_m(\overline{y}_m \cdot \overline{z}_m) \parallel \\ & \textstyle\bigwedge_{i=1}^{m} \neg \exists \overline{y}_i. \ c_i(\overline{x} \cdot \overline{y}_i) \wedge \boldsymbol{f}] \end{aligned}$$

The interesting fact about this kind of definitions is captured by the following lemma:

Lemma 3.2.2. *Given $\{C_1, ..., C_n\}$ a set of exhaustive and nonoverlapping Herbrand constraints and $\{B_1, ..., B_n\}$ a set of first-order formulas with no occurrences of $\overline{x}$, and $\overline{y}_i$ a set of variables included in the free variables of C_i, $i \in \{1 \ldots n\}$ the following holds:*

$$\forall \overline{x}(\neg[\exists \overline{y}_1(C_1 \wedge B_1) \parallel \cdots \parallel \exists \overline{y}_n(C_n \wedge B_n)] \iff$$

$$\iff \exists \overline{y}_1(C_1 \wedge \neg B_1) \parallel \cdots \parallel \exists \overline{y}_n(C_n \wedge \neg B_n))$$

The idea of the transformation to be defined below is to obtain a program whose completion corresponds to the negation of the original program, in particular to a representation of the completion where negative literals have been eliminated. They call this transformation *Negate*. The goal is to produce clauses for a new predicate *neg_p* (for each predicate p in Π) representing its negation.

Definition 3.2.3. *The syntactic transformation negate_rhs is defined as follows:*

$$\begin{aligned} negate\_rhs(P \ ; \ Q) &= negate\_rhs(P) \ , \ negate\_rhs(Q). \\ negate\_rhs(P \ , \ Q) &= negate\_rhs(P) \ ; \ negate\_rhs(Q). \\ negate\_rhs(\boldsymbol{t}) &= \boldsymbol{f}. \\ negate\_rhs(\boldsymbol{f}) &= \boldsymbol{t}. \\ negate\_rhs(p(\overline{t})) &= neg\_p(\overline{t}). \end{aligned}$$

Definition 3.2.4 (Constructive Intensional Negation)**.** *For every predicate p in the original program Π, assuming $def_{\Pi}(p) = \{p(\overline{x}) \ \leftarrow \ B_i(\overline{y}_i \cdot \overline{z}_i) [\!] \ c_i(\overline{x} \cdot \overline{y}_i) \mid i \in 1 \ldots m\}$ a nonoverlapping and exhaustive definition, the negated program ($Negate(\Pi)$) is obtained by adding the following clauses to Π:*

- *If the set of constraints $\{\exists \overline{y}_i.c_i(\overline{x} \cdot \overline{y}_i), i \in \{1..m\}\}$ is not exhaustive, a clause*

$$neg\_p(\overline{x}) \leftarrow \ [\!] \ \bigwedge_{1}^{m} \neg \exists \overline{y}_i.c_i(\overline{x} \cdot \overline{y}_i)$$

- *If $\overline{z}_j$ is empty, the clause* $\quad neg\_p(\overline{x}) \leftarrow negate\_rhs(B_j(\overline{y}_j)) [\!] \ c_j(\overline{x} \cdot \overline{y}_j)$

- *If $\overline{z}_j$ is not empty, the clauses* $neg\_p(\overline{x}) \leftarrow forall(\overline{z}_j,\ p\_j(\overline{y}_j \cdot \overline{z}_j)) \parallel c_j(\overline{x} \cdot \overline{y}_j)$
  $p\_j(\overline{y}_j \cdot \overline{z}_j) \leftarrow negate\_rhs(B_j(\overline{y}_j \cdot \overline{z}_j))$

We can see that negating a clause with free variables introduces "universally quantified" goals by means of a new predicate $forall/2$. This predicate solves them by exploring the Herbrand universe, making use of the following:

1. A universal quantification of the goal Q over a variable X succeeds when Q succeeds without binding (or constraining) X.
2. A universal quantification of Q over X is true if Q is true for all possible values for the variable X.

These two considerations lead to a mutually recursive rule that can be expressed formally as follows:

$$\forall X.\ Q(X) \equiv Q(sk) \vee \left[\forall \overline{X_1}.\ Q(c_1(\overline{X_1})) \wedge \cdots \wedge \forall \overline{X_n}.\ Q(c_n(\overline{X_n}))\right]$$

where $FS_\Sigma = \{c_1 \ldots c_n\}$ is the set of function symbols and sk is a Skolem constant, that is, $sk \notin FS_\Sigma$. In practice, the algorithm proceeds by trying the Skolem case first and, if this fails, then it expands the variables in all possible constructors.

Theoretical soundness and completeness results are given, and the authors show how it is implemented. However, its complexity and the fact that this algorithm does not work in a Prolog way (i.e. returning the solutions one by one) lead us to propose a new theoretical approach to evaluate the "universally quantified" goals, shown in Sec. 3.5.

The method that the authors introduce to translate overlapping rules from general sets of clauses into non-overlapping ones is the following:

Lemma 3.2.3. *Let p be such that*

$$def_\Pi(p) = \{p(\overline{x}) \ \leftarrow \ B_i(\overline{y}_i \ \cdot \ \overline{z}_i) \parallel c_i(\overline{x} \ \cdot \ \overline{y}_i) \mid i \ \in \ 1 \ \ldots \ m\}$$

and there exist $j,k \in 1 \ldots m$ such that $\exists \overline{y}_j.\, c_j(\overline{x} \cdot \overline{y}_j)$ and $\exists \overline{y}_k.\, c_k(\overline{x} \cdot \overline{y}_k)$ are compatible. Then the j-th and k-th clauses can be replaced by the clause

$$p(\overline{x}) \leftarrow B_j(\overline{y}_j \cdot \overline{z}_j); B_k(\overline{y}_k \cdot \overline{z}_k) \parallel c_j(\overline{x} \cdot \overline{y}_j) \wedge c_k(\overline{x} \cdot \overline{y}_k)$$

*and the following additional clauses (if the new constraints are not equivalent to **f**):*

$$p(\overline{x}) \leftarrow B_j(\overline{y}_j \cdot \overline{z}_j) \parallel c_j(\overline{x} \cdot \overline{y}_j) \wedge \neg \exists \overline{y}_k.c_k(\overline{x} \cdot \overline{y}_k)$$
$$p(\overline{x}) \leftarrow B_k(\overline{y}_k \cdot \overline{z}_k) \parallel \neg \exists \overline{y}_j.c_j(\overline{x} \cdot \overline{y}_j) \wedge c_k(\overline{x} \cdot \overline{y}_k)$$

without changing the standard meaning of the program. The process can be repeated if there are more than two overlapping clauses. It is clear that it is a finite process.

3.3 Derivation Procedure for Extended Stable Models

In [PAP+91] the authors show a derivation procedure to determine if a ground query belongs to the well founded model of a ground program. We introduce here the mentioned procedure, starting by the propositions, definitions and theorems needed to understand them and referring the reader to the paper for more details. As a general idea, the authors define the derivation procedure, show the steps to build a WFM-Tree from the derivations and demonstrate that for a program Π and a literal L, if L is in the WFM-Tree of Π then L belongs to the WFM of Π. The soundness and completeness theorems shown in this section are identically applicable to the WFM-derivation modified that we present in Def. 3.4.1.

Definition 3.3.1. *A positive (resp. negative) interpretation I is a set of positive (resp. negative) literals from $Lit(\Pi)$.*

Definition 3.3.2. *A context C_n is an ordered set of positive or negative interpretations. Let S_i be a positive or negative interpretation; C_n denotes the context $S_1 S_2 \ldots S_n$. C_n is a negative (resp. positive) context if S_n is a negative interpretation (resp. positive interpretation). $C_n + G$ denotes the concatenation $S_1 S_2 \ldots S_n G$. A literal G is in context C_n ($G \in C_n$ for short) iff $G \in S_n$. A context C_n implicitly defines an interpretation $I_n(C_n)$ which is the set of literals in partial interpretations S_n i.e. $I_n(C_n) = \cup_{(i \leq n)} S_i$, and for no atom A both A and $\neg A$ belong to it.*

Definition 3.3.3. *A contextual formula (C-formula) is a pair C#F, where C is a context and F is an expression built from atoms with conjunctions and negations. An empty C-formula is the C-formula C#**t**.*

By the interpretation $I(C\#F)$ we mean the interpretation $I(C)$ associated with context C.

Definition 3.3.4 (WFM-derivation)**.** *Let $R_j = < C_j \# F_j \ ; \ I_j >$ where C_j is a context, and I_j a set of literals. A WFM-derivation from R_i to R_n is a sequence from $< C_i \# F_i \ ; \ I_i >$ to $< C_n \# F_n \ ; \ I_n >$ such that for any $< C_k \# F_k \ ; \ I_k >$ $(i \leq k \leq n)$, the following derivation rules apply (where we assume $C_{k+1} = C_k$ and $I_{k+1} = I_k$ unless stated otherwise).*

D.1) if $F_k = \neg G$ and there is no rule $G \leftarrow B$ then $R_{k+1} = (C_k + \neg G \# \mathbf{t} \ ; \ I_k \cup \{\neg G\})$

D.2) if $F_k = \neg G$ and $\neg G \in C_k$ then $F_{k+1} = \mathbf{t}$

D.3) if $F_k = \neg G$ and there are r rules for G with G_i $(1 \leq i \leq r)$ as head, $\neg G \notin C_k$, $G \notin I_k$

$$G_1 \leftarrow B_{11} \ldots B_{m1}$$
$$\ldots$$
$$G_r \leftarrow B_{1r} \ldots B_{m'r}$$

in Π and $G \notin I_k$ then $R_{k+1} = < C_k + \neg G \# \widetilde{G_1}, \ldots, \widetilde{G_r} \ ; \ I_k \cup \{\neg G\} >$ where $\widetilde{G_i}$ is a short hand for $\neg(B_{1i}, \ldots, B_{mi})$

D.4) if $F_k = \neg(G_1, \ldots, G_m)$ then $R_{k+1} = < C_k \# \neg G_i ; I_k >$ for some $1 \leq i \leq m$.

D.5) *if* $F_k = G$ *and* $G \notin I_k$ *then for some rule* $G \leftarrow B_1, \ldots, B_m \in \Pi$,
$R_{k+1} = < C_k + G\#(B_1, \ldots, B_m) \ ; \ I_k \cup G >$.

D.6) *if* $F_k = (g, G)$ *then* $R_{k+1} = < C_k\#G \ ; \ I_{gk} >$ *if there is a derivation from* $< C_k\#g \ ; \ I_k >$ *to* $< L\#\mathbf{t} \ ; \ I_{gk} >$.

There is a WFM-derivation for G in Π *iff there is a sequence from* $< \{\ \}\#G \ ; \ \{\ \} >$ *to* $< L\#\mathbf{t} \ ; \ I >$, *for some I.*

As the authors argue, these rules are intuitive when one recalls the definition of 3-valued model (see Def. 2.1.19): rule 1 establishes the Closed World Assumption (CWA). Rule 2 says that a literal may support itself when proving its falsity. Rule 3 says that for an atom to be interpreted as false it has to be proven false in all definitions for it. Rule 4 says that for a body of a rule to be false it is enough to prove some literal in the body to be false. Rule 5 says that for a literal to be true it is enough to have a rule with all body literals true. Rule 6 says that a conjunction of formulas is true if each element is true. Note that $I_k = \cup_{i \leq n} I_i \cup \{ F_k \}$ if F_k is a literal. This means we do not need to explicitly record I_k at each step k but simply consider all the C_j in $(j \leq k)$.

A derivation from $< \emptyset\#G \ ; \ \emptyset >$ to $< L\#t \ ; \ I >$ may be interpreted as the construction of certain trees to be introduced now. These trees are obtained from the derivation rules with the following in mind: at each derivation step $< C_k\#L_k \ ; \ I_k >$, C_k is the ordered ancestor list of literal L_k which is a node of the tree, and I_k is the set of all literals in the nodes already visited by the derivation procedure. In the following we will omit the special symbol **t**.

Definition 3.3.5 (WFM-Tree)**.** *A WFM-tree for G given program* Π, $WFM(G, \Pi)$ *is a finite tree with root G, such that if N is the literal of a node of the tree then:*

WFM-I) *If N is negative, let* $N = \neg L$ *and:*

- *(a)* *if there are no rules for L then N is a leaf (rule 1)*
- *(b)* *if* $\neg L$ *has an identical ancestor A and all literals in the branch from* $\neg L$ *to A are negative then N is a leaf (rule 2)*
- *(c)* *if there are r rules (clauses) for L in* Π,

$$L_1 \leftarrow B_{11} \ldots B_{k1}$$
$$\ldots$$
$$L_r \leftarrow B_{1r} \ldots B_{k'r}$$

then node $\neg L$ *has r immediate descendents* $\neg B_{j1}, \ldots, \neg B_{hr}$, *each one selected from the body of a different clause (rules 3+4).*

WFM-II) *If N is positive then:*

- *(a)* *If there is a fact N in* Π, *then N is a leaf (rule 5)*
- *(b)* *the n immediate descendents are those literals* $B_1, \ldots, B_n$, *such that a rule* $N \leftarrow B_1, \ldots, B_n$ *exists in* Π *(rule 6)*

By considering all possible choices of rules and literals all WFM-trees are obtained.

Proposition 3.3.1. *For every program and goal G there is a WFM-derivation for G in* Π *iff there is a* $WFM\text{-}Tree(G, \Pi)$.

Lemma 3.3.1. *Given a WFM-tree for G then, for any internal node literal H, if its immediate descendents* $D_1 \ldots D_n$ *are in the well founded model, then H is in the WFM.*

Theorem 3.3.2 (Soundness of WFM-Trees)**.** *Let* Π *be a program and L a literal. If there is a WFM-tree for L then L is in* $WFM(\Pi)$.

Theorem 3.3.3 (Completeness of WFM trees)**.** *Let* Π *be a program and L a literal. If L is in* $WFM(\Pi)$ *there is a WFM-tree for L.*

3.4 The general picture of our implementation

As overviewed in the previous sections, NAF (see Sec. 3.1) is not adequate when dealing with non-ground queries or programs with loops. Clark's Semantics (see Sec. 2.2) or its variants (Fitting or Kunen semantics, see subsection 2.2.1) still have some drawbacks (see subsection 2.2.2) and are not suitable too. The most suitable semantics is the Well Founded Semantics, but the procedures implemented have serious drawbacks as the floundering problem previously exposed in the actual chapter.

Our proposal here is the combination of the symmetrical treatment of the literals done by the Constructive Intensional Negation implementations and the derivation procedure used to determine if the query belongs to the Well Founded Model of the program. By using them we expect the floundering problem to disappear, and we obtain an implementation of Well Founded Semantics able to answer non-ground queries.

As in [MMNMH08], we start from a nonoverlapping set of clauses (see Def. 3.2.2) and apply lemma 3.2.1 to get to a non-overlapping and exhaustive set of clauses from which lemma 3.2.2 is able to obtain the dual of the program.

So, from a non-overlapping set of clauses with the structure

$$def_{\Pi}(p) = \{p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i) [\!] c_i(\overline{x} \cdot \overline{y}_i) \mid i \in 1 \ldots m\}$$

what we obtain is a program with the following structure:

- the positive part (the original program):

$$p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i) [\!] c_i(\overline{x} \cdot \overline{y}_i) \mid i \in 1 \ldots m$$

- and the negative part (the dual program):
  - If the set of constraints $\{\exists \overline{y}_i.\ c_i(\overline{x} \cdot \overline{y}_i), i \in \{1..m\}\}$ is not exhaustive, a clause

$$neg\_p(\overline{x}) \leftarrow [\!] \bigwedge_{1}^{m} \neg \exists \overline{y}_i . c_i(\overline{x} \cdot \overline{y}_i)$$

– For each clause

$$p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i) \parallel c_i(\overline{x} \cdot \overline{y}_i) \mid i \in 1 \ldots m$$

∗ If $\overline{z}_i$ is empty, the clause

$$neg\_p(\overline{x}) \leftarrow negate\_rhs(B_i(\overline{y}_i)) \parallel c_i(\overline{x} \cdot \overline{y}_i)$$

∗ If $\overline{z}_i$ is not empty, the clauses

$$neg\_p(\overline{x}) \leftarrow forall(\overline{z}_i,\ p\_i(\overline{y}_i \cdot \overline{z}_i)) \parallel c_i(\overline{x} \cdot \overline{y}_i)$$
$$p\_i(\overline{y}_i \cdot \overline{z}_i) \leftarrow negate\_rhs(B_i(\overline{y}_i \cdot \overline{z}_i))$$

While the predicate $negate\_rhs/1$ is defined as in [MMNMH08] (see Def. 3.2.3), we propose a new implementation of the predicate $forall/2$, the universal quantification, which is exposed in Sec. 3.5.

This is the point where the derivation procedure for the Well Founded Semantics presented in [PAP+91] (see Sec. 3.3, Def. 3.3.4) enters in action. As we have introduced before, the derivation procedure is defined for non-dual ground programs. Since what we have is a non-ground dual program, we have slightly modified it to fit our purposes.

The first modification is in the derivations of Def. 3.3.4 that take care of negated goals. With the dual program we do not need to assure that the positive rules for the goal can not be derived in order to have a negative derivation, we just need to expand the derivation, by using the rules for the negative goal, in a similar way to what is done with the positive goals. As a result, the derivations D.1, D.3 and D.4 are removed, and D.2 (which now is D.1) only allows negative loops with negative goals in the loop. Derivations D.5 and D.6 are now D.2 and D.4 (they have not been changed) and the new derivation D.3 is added in order to cope with the instantiation of variables.

Definition 3.4.1 (WFM-derivation modified)**.** *Let $R_j = < C_j \# F_j\ ;\ I_j >$ where C_j is a context, and I_j a set of literals. A WFM-derivation from R_i to R_n is a sequence from $< C_i \# F_i\ ;\ I_i >$ to $< C_n \# F_n\ ;\ I_n >$ such that for any $< C_k \# F_k\ ;\ I_k >$ $(i \leq k \leq n)$, the following derivation rules apply (where we assume $C_{k+1} = C_k$ and $I_{k+1} = I_k$ unless stated otherwise).*

Remark. While when we write neg_G we only refer to the negative part of the predicate G, when we write G we refer to both of them. This is done to simplify the derivation procedure.

D.1) if $F_k = neg\_G$, $neg\_G \in C_k$, $C_k = \ \ldots \# neg\_G \# H_1 \# \ldots \# H_n$ and H_i is always a negative goal, then $F_{k+1} = \mathbf{t}$

D.2) if $F_k = G$ and $G \notin I_k$ then for some rule $G \leftarrow B_1, \ldots, B_m \in \Pi$, $R_{k+1} = < C_k + G\#(B_1, \ldots, B_m)\ ;\ I_k \cup G >$.

D.3) if $F_k = G$, $G \notin I_k$ and there is a substitution θ such that $G\theta$ unifies with H, head of a rule $H \leftarrow B_1, \ldots, B_m$, then $R_{k+1} = < C_k + H\#(B_1, \ldots, B_m)\ ;\ I_k \cup H >$.

D.4) if $F_k = (g,\ G)$ *then* $R_{k+1} = < C_k\#G\ ;\ I_{gk} >$ *if there is a derivation from* $< C_k\#g\ ;\ I_k >$ *to* $< L\#\mathbf{t}\ ;\ I_{gk} >$.

There is a WFM-derivation for G in Π *iff there is a sequence from* $< \{\ \}\#G\ ;\ \{\ \} >$ *to* $< L\#\mathbf{t}\ ;\ I >$, *for some I.*

Now we proceed to redefine the way the derivation procedure is interpreted into a WFM-Tree, so that soundness and completeness theorems exposed in section 3.3 can be applied here too.

Definition 3.4.2 (WFM-Tree)**.** *A WFM-tree for G given program* Π, $WFM(G,\Pi)$ *is a finite tree with root G, such that if N is the literal of a node of the tree then:*

WFM-I) If N is negative, let $N = \neg L$ *and:*

- *(a) if* $\neg L$ *has an identical ancestor A and all literals in the branch from* $\neg L$ *to A are negative then N is a leaf (rule D.1).*

WFM-II) No matter if N is positive or negative:

- *(a) if there is a fact N in* Π, *then N is a leaf (rule D.2).*
- *(b) the n immediate descendents are those literals* $B_1, \ldots, B_n$, *such that a rule* $N \leftarrow B_1, \ldots, B_n$ *exists in* Π *(rule D.4).*
- *(c) if there exists a substitution* θ *such that* $N\theta$ *unifies with H and* $H \leftarrow B_1, \ldots, B_n$ *exists in* Π *then the n immediate descendents are those literals* $B_1, \ldots, B_n$ *and the substitution* θ *must be applied to every occurrence of N (rule D.3).*

By considering all possible choices of rules and literals all WFM-trees are obtained.

With these modifications in the derivation procedure, we can determine if a non-ground query belongs or not to the Well Founded Model of the dual program. The pending task is the implementation of the predicate *forall*/2.

3.5 The universal quantification

In this section we expose the existing problem when evaluating the dual of a program that originally had free variables in the body of at least one of its clauses, and the solution that we have adopted. Consider the following program with its dual:

Program 3.5.1

```
p ← q(X).                                  1
q(1).                                      2
                                           3
neg_p ← forall(X, neg_q(X)).               4
neg_q(X) ← X ≠ 1.                          5
```

Remark. The symbol $\neq$ that appears in the programs in this section is a shortcut for the predicate *dist*/2, whose implementation is explained in Sec. 4.4.

Apart from the fact that $q(X)$ and $neg\_q(X)$ are different predicates, there is a big difference in their evaluation. For $q(X)$, it is enough to find a value for X making it true, while $neg\_q(X)$ has to be true for all the values of X. Basically, the first one is an existentially quantified query equivalent to the logic formula "$\exists X.\ q(X)$", while the second one is an universally quantified query equivalent to the logic formula "$\forall X.\ neg\_q(X)$".

Prolog interpreters are based on unification, so if during a proof a free variable needs to be bound they will do it. While this procedure is equivalent to solve the existential quantification, unification is not suitable for solving universally quantified variables: Prolog has no implementation to deduce if the universally quantification holds.

Some $\forall$-implementations have been proposed in [MMNMH08, Sat89, SM91]. In the first one the authors use a method based on a two steps procedure: 1) they test for the variable to be free by using a Skolem constant and 2) if it is not free they test that the formula holds for every possible value for the variable. In the second and the third ones the authors remove the universally quantified implications by transforming them into Prolog programs with disequality constraints.

Both of them result very expensive in terms of computationally costs: the first due to the expansion of code needed in the second step and the second one due to the huge amount of inequalities that obtains from the transformation. So, we have implemented a new version of the universal quantification. The general idea of the method is the following: if we can build a tautology joining by disjunction the different solutions that the variable can take in the formula, then this variable is not constrained and the universal quantification succeeds.

We introduce the method by means of the following program:

Program 3.5.2

```
p ← q(X), neg_q(X).                                    1
q(1).                                                  2
                                                       3
neg_p ← forall(X, (neg_q(X) ; q(X))).                  4
neg_q(X) ← X ≠ 1.                                      5
```

On the one hand, it is obvious that p is not provable, because $q(X)$ and $neg\_q(X)$ can not be true at the same time. On the other hand, if p is not provable then neg_p has to be provable. In order to determine if neg_p is provable or not we apply our method:

1. We obtain the solutions for the variable X in the formulae ($neg\_q(X)$; $q(X)$), which are $X \neq 1$ and $X = 1$.

2. We try to form a tautology by combining the solutions by disjunction. We obtain the tautology $X \neq 1 \vee X = 1$.

3. As we have built a tautology, the variable is not constrained. So, $forall(X,\ (neg\_q(X)\ ;\ q(X))\ )$ succeeds.

The algorithm used is presented below:

Algorithm 3.5.1 Algorithm to determine if the universal quantification succeeds

1. Obtain the different solutions that the variable can take in the formulae. (It does not matter if we get all the solutions in one step or we get some of them and backtrack to obtain more if they were not enough).
2. (a) If in one solution the variable is free (it is not bound or constrained) the forall predicate succeeds.
   (b) If not, we take a constrained value (or a constant value) for the variable and look for the necessary constants (or constrained values) to build a tautology and free the variable. If the tautology can be built the forall succeeds.

We illustrate how the algorithm behaves in the following examples.

Example 3.5.1 In this example we show the original program, how it becomes after adding to it the negated counterpart of the predicates and how the $\forall$ implementation looks for the tautology it needs to free the variable X.

Program 3.5.3 Original program

```
q(X) ← mbr2(X, [1,2,3]).                                    1
r(X) ← q(X) ; neg_q(X).                                     2
s ← neg_r(X).                                               3
```

Program 3.5.4 Translated program

```
q(X) ← mbr2(X, [1,2,3]).                                    1
neg_q(X) ← neg_mbr2(X, [1,2,3]).                            2
                                                            3
r(X) ← q(X) ∨ neg_q(X).                                     4
neg_r(X) ← neg_q(X) ∧ q(X).                                 5
                                                            6
s ← neg_r(X).                                               7
neg_s ← ∀ X. r(X).                                          8
```

The query in which we are interested here is $? - neg\_s$. To answer it, we first obtain the values for the variable X in the query $r(X)$ that prove it. The valid values for X are $X = 1$, $X = 2$, $X = 3$ and $X \neq 1 \wedge X \neq 2 \wedge X \neq 3$, as can be seen below.

Evaluation of query $r(X)$ in program 3.5.4

```
?- r(X).
   X = 1;
   X = 2;
   X = 3;
   X ≠ 1 ∧ X ≠ 2 ∧ X ≠ 3;
   no
```

Once we have the solutions or valid values for X, we joint them by disjunction: X = 1 $\vee$ X = 2 $\vee$ X = 3 $\vee$ (X $\neq$ 1 $\wedge$ X $\neq$ 2 $\wedge$ X $\neq$ 3). After that we try to solve the disjunction: X $\neq$ 1 is removed by using X = 1, X $\neq$ 2 is removed by using X = 2 and X $\neq$ 3 is removed by using X = 3. As we have not introduced any new constraint for the variable X and all the initial constraints have been removed, the variable X is free. As X is free, then $\forall X.\ r(X)$ succeeds and finally $?- neg\_s$ succeeds.

Example 3.5.2 In this example the variable is free because there exists a rule in the original program that does not make any restriction on the variable. It can be seen equivalent to the evaluation of the program with a Skolem constant that is presented in [MMNMH08], but it is much simpler: If the variable is not bound or constrained, the $\forall$-quantification succeeds.

Program 3.5.5 Original program

```
r(X).                                                    1
s ← neg_r(X).                                            2
```

Remark. When the dual of a clause can never succeed, it is removed from the final program. This is the reason why $neg\_r(X)$ has no rules.

Program 3.5.6 Translated program

```
r(X).                                                    1
s ← neg_r(X).                                            2
neg_s ← forall(X, r(X)).                                 3
```

The query in which we are interested here is $?- neg\_s.$, and to answer it our implementation retrieves all the answers for $r(X)$. In this case, X is not bound or constrained, so the $\forall$-quantification succeeds without looking for a tautology (or assuming that one of the answers is a tautology itself).

CHAPTER 4

THE IMPLEMENTATION DETAILS

In the previous chapter we have revised the existing implementations of Well Founded Semantics, and it was introduced the theory that supports our implementation: the calculation of the dual (Sec. 3.4), the derivation procedure that allows to get results according to the Well Founded Model (Sec. 3.4) and the evaluation of the universal quantification (Sec. 3.5). In this section we show what is behind the scenes, we expose the details of the implementation.

We start by describing the method that assures that our input program has only nonoverlapping clauses (Sec. 4.1), something that is a necessary condition for applying lemmas 3.2.1 and 3.2.2. Next we describe how these lemmas are applied to obtain the negated counterpart of the program (Sec. 4.2) and, finally, the obtained code is modified to obtain the Well Founded Semantics results instead of the CLP version of Kunen Semantics that the authors of [MMNMH08] obtain (Sec. 4.3). After that, we present the implementation of inequalities via attributed variables (Sec. 4.4) and the *forall*/2 predicate (Sec.4.5), both needed for the whole implementation to work.

Remark. The algorithms and procedures presented in the following sections take as synonymous input program and set of clauses with the same head's functor, but they are obviously not (In a program we can have several sets of clauses with different head's functors). This is due to the fact that, before any other task, the method presented here splits the input program into sets of clauses with the same head's functor, and deals with each set separately. It is done in this way for the sake of simplicity, but (in case it is necessary) it is easy to generalize the method for the whole input program.

4.1 Transform Overlapping Clauses into Nonoverlapping Ones

When programming in Prolog, we usually do it by cases. For example, when implementing *Peano numbers* (See program 4.1.1) we use two clauses, one for the basic case and the second one for recursion.

Program 4.1.1 Peano example

```
p(0).                                                    1
p(s(X)) ← p(X).                                          2
```

Although this is how Prolog programmers try to code, sometimes it is not possible to do it in that way, and overlapping clauses (see Def. 3.2.2) appear in programs. In program 4.1.2 the second and the third clauses are overlapping, both of them can be used to directly evaluate the query

$$? - prettyPrinting([' Here\ we\ are!!']).$$

Program 4.1.2 Pretty printing example

```
prettyPrinting([]).                                      1
prettyPrinting([Sentence]) ←                             2
        prettyPrintingSentence(Sentence).                3
prettyPrinting([Sentence|Others]) ←                      4
        prettyPrintingSentence(Sentence),                5
        nl,                                              6
        prettyPrinting(Others).                          7
```

As lemmas 3.2.1 and 3.2.2 rely on the fact that our program has only nonoverlapping clauses, we must turn a program with overlapping clauses into an equivalent one without overlapping clauses. In [MMNMH08] this is done as described in algorithm 4.1.1.

Algorithm 4.1.1 Algorithm to transform overlapping clauses into nonoverlapping ones (from [MMNMH08])

1. Input: *Clauses_Set* [Set of clauses of the original program].
2. While there are two overlapping clauses in *Clauses_Set*, loop.
   - 2.1. Name the overlapping clauses Cl_1 and Cl_2, and remove them from the input set.
   - 2.2. Build a new clause $Cl\_3 := (\ Head \leftarrow Body\ )$.
     - $Head := overlap(Cl\_1,\ Cl\_2)$. [a] [b]
     - $Body := or(body(Cl\_1),\ body(Cl\_2))$. [c]
   - 2.3. Remove the overlap from each one of the two selected clauses.
     - If the result from removing it is an unsatisfiable clause then forget about this clause.
     - If not, add the resultant clause to the set of clauses.
3. Output: *Clauses_Set* [Now non-overlapping clauses].

[a] The symbol ":=" is used to assign to the variable to the left the value to the right.
[b] The head of this clause is the overlap between Cl_1 and Cl_2.
[c] The body of this clause is the result of joining with the operation 'or' their bodies.

Apart from the fact that algorithm 4.1.1 can expand the program to a large amount of clauses (up to a maximum of n+(n/2) clauses if the original program has n clauses), it is very expensive computationally speaking (each time they remove one overlap between two clauses they have to test again the whole set of clauses in order to determine if there is another overlap between two of them). Besides, no method is presented to evaluate if two clauses are compatible (if there is an overlap between them) or not.

We present a new approach that removes this problems by finding all the overlappings between the clauses in one step (by means of evaluating all the possible conjunctions between the clauses' heads, see algorithms 4.1.2 and 4.1.3) and transforming all the involved clauses into nonoverlapping clauses in another step (see algorithm 4.1.4). We start by generalizing the overlap between only two clauses to two or more clauses.

Definition 4.1.1 (Overlapping between two or more clauses)**.** *Let p be such that*

$$def_{\Pi}(p) = \{\ Cl_i \mid i\ \in\ 1\ \ldots\ m\ \}$$

$$Cl_i = \{\ p(\overline{x})\ \leftarrow\ B_i(\overline{y}_i\ \cdot\ \overline{z}_i) [\!] c_i(\overline{x}\ \cdot\ \overline{y}_i)\ \}$$

so our set of clauses for the predicate p is $\{Cl_1,\ Cl_2,\ \ldots,\ Cl_m\}$.
There is an overlapping between n clauses, $n \leq m$ and $n \geq 2$, if the conjunction of their constraints is compatible, if

$$\bigwedge_{1}^{n}\ \exists y_i.\ c_i(\overline{x} \cdot \overline{y}_i)$$

is satisfiable.

Lemma 4.1.1 (Overlapping clauses to Nonoverlapping ones)**.** *Let p be such that*

$$def_{\Pi}(p) = \{\ Cl_i \mid i\ \in\ 1\ \ldots\ m\ \}$$

$$Cl_i = \{\ p(\overline{x})\ \leftarrow\ B_i(\overline{y}_i\ \cdot\ \overline{z}_i) [\!] c_i(\overline{x}\ \cdot\ \overline{y}_i)\ \}$$

so our set of clauses for the predicate p is $\{Cl_1,\ Cl_2,\ \ldots,\ Cl_m\}$. If there exists a subset of this set, say $\{\ Cl_{k1},\ Cl_{k2},\ \ldots,\ Cl_{kn}\}$, such that

$$\bigwedge_{i=1}^{i=n}\ \exists y_{ki}.\ c_{ki}(\overline{x} \cdot \overline{y}_{ki})$$

is satisfiable, then this clauses can be replaced by the clause

$$p(\overline{x}) \leftarrow \bigvee_{i=1}^{i=n}\ B_{ki}(\overline{y}_{ki} \cdot \overline{z}_{ki})\ [\!]\ \bigwedge_{i=1}^{i=n}\ \exists y_{ki}.\ c_{ki}(\overline{x} \cdot \overline{y}_{ki})$$

*and the following additional clauses (if the new constraints are not equivalent to **f**):*

$$p(\overline{x}) \leftarrow B_j(\overline{y}_j \cdot \overline{z}_j)\ [\!]\ c_j(\overline{x} \cdot \overline{y}_j)\ \wedge\ \neg \bigwedge_{1}^{n}\ \exists y_i.\ c_i(\overline{x} \cdot \overline{y}_i)$$

$$p(\overline{x}) \leftarrow B_k(\overline{y}_k \cdot \overline{z}_k)\ [\!]\ c_k(\overline{x} \cdot \overline{y}_k)\ \wedge\ \neg \bigwedge_{1}^{n}\ \exists y_i.\ c_i(\overline{x} \cdot \overline{y}_i)$$

without changing the standard meaning of the program.

The following algorithms 4.1.2 and 4.1.3 find in the input program all the satisfiable conjunctions between the clauses' heads.

Algorithm 4.1.2 Find conjunctions between the clauses' heads (I)

1. Input: (Clauses) [List of clauses of the original program].
2. Initialize an empty list called Conjunctions.
3. Clauses_1 := Clauses.
4. while the list Clauses_1 has more than one clause, loop[a]
   - 4.1. Clauses_2 := tail(Clauses_1)[b]
     Clauses_3 := tail(Clauses_2)
     First := head(Clauses_1)
     Second := head(Clauses_2)
   - 4.2. Search for a conjunction between First and Second.
     - 4.2.i. If there is a conjunction between them, this one will be Conj_1.
       1. Initial_Solution := [First, Second][c]
       2. Run algorithm 4.1.3 with this arguments: (Conj_1, Clauses_3, Initial_Solution).
       3. Append the elements in the returned list to the Conjunctions list.
     - 4.2.ii. If not, continue.
   - 4.3. Clauses_1 := Clauses_2.
5. Output: (Conjunctions) [The list of conjunctions between the clauses' heads.]

[a] If Clauses_1 is empty or has only one clause, there are no more conjunctions.
[b] The functions head and tail obtain respectively the head and the tail of a list
[c] This notation means that Initial_Solution is a list composed by the elements First and Second. The head of the list is First.

Algorithm 4.1.3 Find conjunctions between the clauses' heads (II)

1. Input: (Conj_1, Clauses_4, Initial_Solution) [Clauses_4 and Initial_Solution are lists of clauses]
2. Initialize an empty list, Current_Solution.
3. If Clauses_4 is empty.
   - 3.1. Append "conjunction(Conj_1, Initial_Solution)" to the list Current_Solution.
4. If Clauses_4 is not empty.
   - 4.1. Third := head(Clauses_4)
     Clauses_5 := tail(Clauses_4)
   - 4.2. Run algorithm 4.1.3 with this arguments:
     (Conj_1, Clauses_5, Initial_Solution).
   - 4.3. Append the elements in the returned list to the Current_Solution list.
   - 4.4. Search for a conjunction between Conj_1 and Third.
     - 4.4.i. If there is a conjunction between them, this one will be Conj_2.
       1 New_Initial_Solution is the result of append Third to the end of the list Initial_Solution.
       2 Run algorithm 4.1.3 with this arguments:
       (Conj_2, Clauses_5, New_Initial_Solution).
       3 Append the elements in the returned list to the Current_Solution list.
     - 4.4.ii. If not, continue.
5. Output: (Current_Solution) [The list with the current solutions.]

The algorithm searches for all possible conjunctions between the heads of the set of clauses given and returns a list of elements "$conjunction(Conj,\ Affected\_Clauses)$". In this list we may have results contained into other results. The cause is that if we have, for example, c as the conjunction for the clauses 1, 2 and 3 then it will try to find the conjunction between 1 and 2, 1 and 3 and 1, 2 and 3. So, we will obtain as solution the following list:

$$[\ conjunction(c,[1,2,3]), conjunction(c,[1,2])conjunction(c,[1,3])\ ]$$

As we are only interested in the big one, determining which solution is the big one is trivial. For simplicity, this procedure is not included here.

The following algorithm is in charge of converting the overlapping clauses into nonoverlapping ones, by using the information provided by the previous algorithms.

Algorithm 4.1.4 Algorithm to transform overlapping clauses into nonoverlapping ones

1. Input: (Clauses, Conjunctions) [Clauses with same head's functor and set of conjunctions between them.]
2. Initialize an empty list, Non-Overlappings.
   List_1 := Conjunctions
3. While List_1 is not empty, loop
   3.1. First_1 := head
   Others_1 := tail(List_1)
   3.2. The clauses Conj_Clause is formed as follows:
   - Its head is the conjunction in the variable First (the variable First contains an element "conjunction(Conj, Affected_Clauses)").
   - Its body is formed by the disjunction of the bodies of the affected clauses, which are again in the variable First.

   3.3. Append Conj_Clause to the list Non-Overlappings
   3.4. List_1 := Others_1
4. List_2 := Clauses
5. while List_2 is not empty, loop
   5.1. First_2 := head(List_2)
   Others_2 := tail(List_2)
   5.2. For each conjunction in which First_2 appear, the clause First_2_Non_Overlap is formed as follows:
   - The head of the new clause is the old one removing from it the conjunction.
   - The body of the new clause is the old one.

   5.3. Append First_2_Non_Overlap to the list Non-Overlappings.
   5.4. List_2 := Others_2
6. Output: Non-Overlappings (Set of nonoverlapping clauses).

The improvements, compared to [MMNMH08], are the following:

- instead of looking for an overlap between two clauses we look for the conjunctions between all of them,
- instead of taking only two overlapping clauses at the same time we take all the clauses with a common conjunction,
- instead of joining the bodies of two clauses we joint the bodies of all affected clauses,
- instead of removing the existing overlap between two clauses we remove the set of accepted values by the conjunction from all the affected clauses and
- where it tests again the whole set for another overlap between two clauses we proceed to remove another conjunction (we do not need to test again).

Obviously this procedure saves computational time and, as it generates only one clause for the existing conjunction between two or more clauses instead of one clause for each overlap between two clauses, it generates more compact and simple programs.

4.2 Calculating the Negation of the clauses

After assuring that our input program has only nonoverlapping clauses, we proceed to make the completion of the program, so finally we can take from it the negative part we are looking for. The method introduced in [MMNMH08] relies on lemmas 3.2.1 and 3.2.2, presented in section 3.2. Here we first show how to make a practical use of them and, only after that, how to implement them.

As introduced before, the procedure in [MMNMH08] relies on the exhaustiveness of the clauses, a prerequisite achieved by means of lemma 3.2.1. This lemma converts a non-exhaustive set of clauses into an exhaustive one by introducing a new rule in the program (its last rule). The resulting exhaustive set of clauses is then negated clause by clause, as exposed in lemma 3.2.2, obtaining finally the dual of the input program.

Lemma 3.2.1 says that if our program is composed by a set of nonoverlapping and non-exhaustive clauses, its definition is the following:

$$def_{\Pi}(p) = \{p(\overline{x}) \leftarrow B_i(\overline{y}_i \cdot \overline{z}_i) \parallel c_i(\overline{x} \cdot \overline{y}_i) \mid i \in 1 \ldots m\}$$

and then its completed definition is logically equivalent to

$$\begin{aligned} cdef_{\Pi}(p) \equiv \forall \overline{x}.[p(\overline{x}) \iff\ & \exists \overline{y}_1.\ c_1(\overline{x} \cdot \overline{y}_1) \wedge \exists \overline{z}_1.B_1(\overline{y}_1 \cdot \overline{z}_1) \parallel \\ & \vdots \\ & \exists \overline{y}_m.\ c_m(\overline{x} \cdot \overline{y}_m) \wedge \exists \overline{z}_m.B_m(\overline{y}_m \cdot \overline{z}_m) \parallel \\ & \textstyle\bigwedge_{i=1}^{m} \neg \exists \overline{y}_i.\ c_i(\overline{x} \cdot \overline{y}_i) \wedge \mathbf{f}], \end{aligned}$$

where the last clause is added in case that the previous definition of the predicate is not exhaustive. We will see how this is implemented in practice by means of the example in program 4.2.1.

Program 4.2.1 *bigger_than* **predicate in Peano numbers**

```
bt(s(X), s(Y)) ← bt(X, Y)                                    1
bt(s(X), 0).                                                 2
```

For the code of the previous program (in the example there are no overlapping clauses, but if we had overlapping clauses we should apply before the procedure presented in section 4.1) its translation in logic is the following:

$$cdef_{\Pi}(bt) \equiv \forall \overline{x}.[bt(\overline{x}) \iff \begin{array}{l} \exists \overline{y}_1.\ c_1(\overline{x} \cdot \overline{y}_1) \wedge \exists \overline{z}_1.B_1(\overline{y}_1 \cdot \overline{z}_1) \parallel \\ \exists \overline{y}_2.\ c_2(\overline{x} \cdot \overline{y}_2) \wedge \exists \overline{z}_2.B_2(\overline{y}_2 \cdot \overline{z}_2) \end{array}$$

where

$$\begin{array}{rl} c_1(\overline{x} \cdot \overline{y}_1)\ is & (\overline{x} == (s([\overline{y}_1]_1),\ s([\overline{y}_1]_2)) \\ c_2(\overline{x} \cdot \overline{y}_2)\ is & (\overline{x} == (s([\overline{y}_2]_1),\ 0) \\ B_1(\overline{y}_1 \cdot \overline{z}_1)\ is & bt(\overline{y}_1) \\ B_2(\overline{y}_2 \cdot \overline{z}_2)\ is & \mathbf{t}. \end{array}$$

If we apply lemma 3.2.1, the last clause should be

$$\bigwedge_{i=1}^{i=2} \neg \exists \overline{y}_i.\ c_i(\overline{x} \cdot \overline{y}_i) \wedge \mathbf{f}.$$

Although the expected result from this procedure is the following program,

Program 4.2.2 *bigger_than* **predicate for Peano numbers (exhaustive)**

```
bt(s(X), s(Y)) ← bt(X, Y).                                   1
bt(s(X), 0).                                                 2
bt(0, s(Y)) ← fail.                                          3
```

this is not the result even when we apply lemma 3.2.1. In fact we have taken a decision that the lemma does not support: we know that the clause that is missing is the one that deals with a second argument bigger that the first one ($0 < s(Y)$) because the program that we are developing deals with *Peano numbers* and there are no more options. This is not the correct way of applying lemma 3.2.1.

Lemma 3.2.1 does not say *"look for the missing values"*, but *"add the clause"*

$$\bigwedge_{i=1}^{i=2} \neg \exists \overline{y}_i.\ c_i(\overline{x} \cdot \overline{y}_i) \wedge \mathbf{f}.$$

which again needs to be modified in order to be implemented. The problem here is the same that motivated us for this work: formulas in which existentially quantified variables are negated.

Although we could just apply again the whole procedure developed here, the formulas to which negation is applied are different from the original ones: here the

formulas involved are only formed by conjunction of equalities or disequalities, i.e. $c_i(\overline{x} \cdot \overline{y}_i)$ is always an equality or a disequality.

As a result, we can use logic to convert "$\neg\exists$" into "$\forall\neg$", which is worth for our purposes. The result is the following:

$$\bigwedge_{i=1}^{i=2} \forall \overline{y}_i.\neg\ [\ \ c_i(\overline{x} \cdot \overline{y}_i)\ ] \wedge \mathbf{f}.$$

Although it seems to introduce the universal quantification problem again, the problem here is not as general as the one in Sec. 3.5, and so the solution is much simpler that the one exposed there. We will see how this problem is solved by means of the example previously introduced.

From the translation of the previous example (in which we have substituted $c_i(\overline{x} \cdot \overline{y}_i)$ and $B_i(\overline{y}_i \cdot \overline{z}_i)$ for every i and $\neg\exists$ by $\forall\neg$) we have the following completed definition of our program Π:

$$\begin{array}{rl} cdef_{\Pi}(bt) \equiv \forall \overline{x}.[bt(\overline{x}) \iff & \exists \overline{y}_1.\ \overline{x} == (s([\overline{y}_1]_1),\ s([\overline{y}_1]_2)) \wedge \exists \overline{z}_1.\ bt(\overline{y}_1)\ \| \\ & \exists \overline{y}_2.\ \overline{x} == (s([\overline{y}_2]_1),\ 0) \wedge \exists \overline{z}_2.\ \mathbf{t}\ \| \\ & \\ & \forall \overline{y}_1.\neg\ [\ \ \overline{x} == (s([\overline{y}_1]_1),\ s([\overline{y}_1]_2))\ ]\ \wedge \\ & \forall \overline{y}_2.\neg\ [\ \ \overline{x} == (s([\overline{y}_2]_1),\ 0)\ ] \\ & \wedge\ \mathbf{f}] \end{array}$$

which is equivalent to:

$$\begin{array}{rl} cdef_{\Pi}(bt) \equiv \forall \overline{x}.[bt(\overline{x}) \iff & \exists \overline{y}_1.\ \overline{x} == (s([\overline{y}_1]_1),\ s([\overline{y}_1]_2)) \wedge \exists \overline{z}_1.\ bt(\overline{y}_1)\ \| \\ & \exists \overline{y}_2.\ \overline{x} == (s([\overline{y}_2]_1),\ 0) \wedge \exists \overline{z}_2.\ \mathbf{t}\ \| \\ & \\ & \forall \overline{y}_1.\ \ [\ \ \overline{x} \neq (s([\overline{y}_1]_1),\ s([\overline{y}_1]_2))\ ]\ \wedge \\ & \forall \overline{y}_2.\ \ [\ \ \overline{x} \neq (s([\overline{y}_2]_1),\ 0)\ ] \\ & \wedge\ \mathbf{f}] \end{array}$$

The point of this result is that the last clause can be seen in two ways:

- either we have to test for every possible values of y_1 and y_2 the formula

$$[\ \overline{x} \neq (s([\overline{y}_1]_1),\ s([\overline{y}_1]_2))\ ]\ \bigwedge\ [\ \overline{x} \neq (s([\overline{y}_2]_1),\ 0)\ ]$$

- or we have to whether test that x_1 and x_2 are different from every term built with the structure exposed in the formulas, no matter which values y_1 and y_2 take.

The second one is the option selected. The last definition of our program is then converted into:

$$\begin{array}{rl} cdef_{\Pi}(bt) \equiv \forall \overline{x}.[bt(\overline{x}) \iff & \exists \overline{y}_1.\ \overline{x} == (s([\overline{y}_1]_1),\ s([\overline{y}_1]_2)) \wedge \exists \overline{z}_1.\ bt(\overline{y}_1)\ \| \\ & \exists \overline{y}_2.\ \overline{x} == (s([\overline{y}_2]_1),\ 0) \wedge \exists \overline{z}_2.\ \mathbf{t}\ \| \\ & \\ & [\ \overline{x} \neq (s(\text{"}anything\text{"}),\ s(\text{"}anything\text{"}))\ ]\ \wedge \\ & [\ \overline{x} \neq (s(\text{"}anything\text{"}),\ 0)\ ] \\ & \wedge\ \mathbf{f}]. \end{array}$$

Remark. As this "anything" can not be coded in programs, we use the predicate $fA(\_)$ as a shortcut for "anything". When used, for example in $(V \neq s(fA(\_)))$, it means that V has to be different from the functor $s/1$ applied to anything.

The big difference between using the second option or the first one is that we remove the universal quantification by using a special term $fA(\_)$, which is a meta-term that we handle by means of the inequalities implementation (see Sec. 4.4). The general idea is that, when testing an inequality like $(V \neq s(fA(\_)))$, we just forbid the variable V to be bound to a term formed with the functor $s/1$, which is just what the logic formula $\forall Y.\ (V \neq s(Y))$ means. The resulting code from applying the whole method to the previous example is the following:

Program 4.2.3 *bigger_than* **predicate in Peano numbers (exhaustive)**

```
bt(s(X), s(Y)) ← bt(X, Y)                                          1
bt(s(X), 0).                                                       2
bt(V, W) ← ((V, W) ≠ (s(fA(_)), s(fA(_)))),                        3
           ((V, W) ≠ (s(fA(_)), 0)),                               4
           fail.                                                   5
```

Remark. The predicate $\neq /2$ is not a native predicate. It is in practice implemented by the predicate *dist*$/2$, which uses attributed variables (see Def. 4.4.1) to assure that the variable involved never takes the forbidden value. More details on this can be found in Sec. 4.4.

After applying lemma 3.2.1 to achieve exhaustiveness in the set of clauses we apply lemma 3.2.2 to finally obtain the dual of our input set of clauses. Lemma 3.2.2 says: Given $\{C_1, ..., C_n\}$ a set of exhaustive and nonoverlapping Herbrand constraints and $\{B_1, ..., B_n\}$ a set of first-order formulas with no occurrences of $\overline{x}$, and $\overline{y}_i$ a set of variables included in the free variables of C_i, $i \in \{1 \ldots n\}$ the following holds:

$$\forall \overline{x}(\neg[\exists \overline{y}_1(C_1 \wedge B_1) \parallel \cdots \parallel \exists \overline{y}_n(C_n \wedge B_n)] \iff$$

$$\iff \exists \overline{y}_1(C_1 \wedge \neg B_1) \parallel \cdots \parallel \exists \overline{y}_n(C_n \wedge \neg B_n))$$

So we can compute the dual of the program just by computing separately the negative clause for each one of its clauses. The final result for our example program (the dual program) is shown below.

Program 4.2.4 dual of predicate *bigger_than* **in Peano numbers**

```
neg_bt(s(X), s(Y)) ← neg_bt(X, Y)                                  1
neg_bt(s(X), 0) ← fail.                                            2
neg_bt(V, W) ← ((V, W) ≠ (s(fA(_)), s(fA(_)))),                    3
               ((V, W) ≠ (s(fA(_)), 0)).                           4
```

The algorithm for computing the negative part of a program is shown below.

Algorithm 4.2.1 Algorithm to calculate the negative part of a set of clauses with the same head's functor

1. Input: (Clauses) [Nonoverlapping set of clauses with same head's functor.]
2. Initialize an empty list, Negated_Clauses.
3. Initialize a clause: Negate_Heads_Clause[a].
   - Its head is the head's functor negated[b] (there is only one).
   - Its body is empty.
4. While Clauses is not empty, loop
   4.1. First := head(Clauses)
   Others = tail(Clauses)
   4.2. The variable First contains a clause whose structure is:
   $p(\overline{x}) \leftarrow \exists \overline{y}_N.\ c_N(\overline{x} \cdot \overline{y}_N) \parallel B_N(\overline{y}_N, \overline{z}_N)$
   4.3. Negate $[\ \exists \overline{y}_N.\ c_N(\overline{x} \cdot \overline{y}_N)\ ]$ to obtain $[\ \forall \overline{y}_N. \neg\ c_N(\overline{x} \cdot \overline{y}_N)\ ]$, and joint it via conjunction to the body of Negate_Heads_Clause. If the body was still empty, just take this formula as its new body.
   4.4. The negation of the clause First is Negated_First, whose structure is $\neg p(\overline{x}) \leftarrow \exists \overline{y}_N.\ c_N(\overline{x} \cdot \overline{y}_N) \parallel negate\_rhs(B_N(\overline{y}_N, \overline{z}_N))$[c]
   4.5. Append Negated_First to Negated_Clauses.
   4.6. Clauses := Others
5. Append Negate_Heads_Clause to Negated_Clauses.
6. Output: (Negated_Clauses) [Negated clauses of the set of nonoverlapping clauses with the same head's functor].

[a] This clause will contain the negation of the clause that ensures exhaustiveness.
[b] The negation $\neg p(\bar{t})$ will be coded as $neg\_p(\bar{t})$.
[c] *negate_rhs* is a predicate in charge of evaluating the negation of the right hand side of a clause. It is a syntactic transformation, and the way it works is in Def. 4.2.1.

Definition 4.2.1. *The syntactic transformation negate_rhs is defined as follows:*

```
negate_rhs( (P ; Q) ) = negate_rhs( P ) , negate_rhs( Q ).
negate_rhs( (P , Q) ) = negate_rhs( P ) ; negate_rhs( Q ).
negate_rhs( ( t ) ) = f.
negate_rhs( ( f ) ) = t.
negate_rhs( ( p(t) ) ) = ¬ p(t).
```

where $\mathbf{t}$ *is true,* $\mathbf{f}$ *is false and the negation* $\neg p(\bar{t})$ *will be coded as* $neg\_p(\bar{t})$.

4.3 Adding the well founded semantics behavior

In sections 4.1 and 4.2 we have shown how to obtain the dual part of a program. As the semantics of the resultant program is theoretically based on the procedure presented in CIN [MMNMH08] (see Sec. 3.2), the answers obtained from querying it still obey the CLP version of Kunen Semantics that has been introduced in Sec. 3.2.

As our intended semantics is the Well Founded Semantics, in this section we show how to modify the dual program obtained in Sec. 4.2 to obey them. Although the WFS definition provided in Def. 2.5.1 is said to be symmetrical, implementations (like SLS, SLG, SLT, see chapter 3) usually do not work in this way. To manage the clauses in a symmetrical way it is required to obtain, in some way, the dual of the program (for example via the procedure we introduced in sections 4.1 and 4.2), and none of them do that. We remark that this is the first working implementation that makes use of the dual program to achieve non-ground queries in Well Founded Semantics.

We will first introduce which are the results we want to obtain and, later, how to modify the code obtained from Sec. 4.2 so we can do that. The expected results when making a query to the interpreter is the 3-valued interpretation of this query according to the Well Founded Model of our program, i.e.

- If we get a result for a query G, then the query G belongs to the Well Founded Model (WFM) of the program Π.
- If we fail getting a result then the query G does not belong to the WFM of the program. This does **not** mean that $\neg G \in WFM(\Pi)$.

Remark. From here onwards, we use "$\neg L$" to refer to the negation of a literal L. As now we treat positive and negative literals symmetrically, this means that, for a positive literal L, $\neg L = neg\_L$ and, for a negative literal L, $\neg neg\_L = L$.

To completely assign a (3-valued) truth value to a literal L we need to do the following:

- If we make a query "$? - L.$" and it gets an answer "yes" then L is true and neg_L is false.
- If we make a query "$? - neg\_L.$" and it gets an answer "yes" then neg_L is true and L is false.
- If both queries "$? - L.$" and "$? - neg\_L.$" get the answer "*no*" then the atom is "*undefined*".

Example Ex. 4.3.1 is introduced in order to clarify this: while in WFS the answers for both queries $q(a)$ and $not(q(a))$ should be *undefined*, our answer for them is in both queries *no*. The meaning of this is that none of them belongs to the Well Founded Model of the program Π, so we can conclude that the atom $q(a)$ is *undefined*.

Example 4.3.1 Program and results that we want to obtain by using the introduced semantics

Program 4.3.1

```
p(a) ← p(a).                                    1
neg_p(a) ← neg_p(a).                            2
                                                3
q(a) ← neg_r(a).                                4
neg_q(a) ← r(a).                                5
                                                6
r(a) ← neg_q(a).                                7
neg_r(a) ← q(a).                                8
                                                9
s(a) ← t(a).                                    10
neg_s(a) ← neg_t(a).                            11
                                                12
t(a) ← s(a).                                    13
neg_t(a) ← neg_s(a).                            14
```

Evaluation of queries $p(a)$, $neg\_p(a)$, $s(a)$, $neg\_s(a)$, $q(a)$ and $\neg q(a)$ in program 4.3.1

```
?- p(a).
   no
?- neg_p(a).
   yes
?- s(a).
   no
?- neg_s(a).
   yes
?- q(a).
   no    [undefined]
?- neg_q(a).
   no    [undefined]
```

If we compare implementations from WFS and from the CLP version of KS we find that there are some procedures to obtain solutions in each of them that share most part of the process. In fact, the only difference between them is the management of loops, and the answer given to the user when a loop is found. The result that we want to obtain, and that is in accordance with [PAP+91], when the query contains a goal G involved in a loop is the following:

- If G is a positive literal then it should fail. G $\notin$ WFM(Π). It does not matter if we have a negative literal involved in the loop (undefined in WFS) or not (false in WFS, $\neg$G $\in$ WFM(Π)).
- If G is a negative literal and there are some positive literals involved the loop (undefined in WFS) then it should fail. G $\notin$ WFM(Π).
- If G is a negative literal and there are no positive literals involved in the loop then it should succeed (true in WFS). G $\in$ WFM(Π).

In order to implement this behavior we need to track the execution and examine the derivation that has been performed when a loop is found. For doing that we use

a memorizing technique, which basically stores the whole derivation and allows us to examine it when a loop is found. The implementation obtained from Sec. 4.2 (for each predicate) is modified in the following way:

- We change the name of each one of the clauses not in the dual program from

  $$p(\overline{x}) \leftarrow body. \qquad \text{to} \qquad positive\_p(\overline{x}) \leftarrow body.$$

  and we add a new clause in charge of examining that there are no positive loops,

  $$p(\overline{x}) \leftarrow test\_no\_loops\_on(p(\overline{x})) \bigwedge positive\_p(\overline{x}).$$

- We change the name of each one of the clauses in the dual program from

  $$neg\_p(\overline{x}) \leftarrow body. \qquad \text{to} \qquad negative\_p(\overline{x}) \leftarrow body.$$

  and we add two new clauses, the first in charge of testing that we have no loops (so we can continue executing the old body of $neg\_p(\overline{x})$, $negative\_p(\overline{x})$) and the second one to make the predicate succeed when there is a loop and only negative literals are involved in the loop. Making the predicate fail when there are some positive literal involved in the loop does not need to be coded.

  $$\begin{array}{ll} neg\_p(\overline{x}) \leftarrow & test\_no\_loops\_on(neg\_p(\overline{x})) \bigwedge negative\_p(\overline{x}). \\ neg\_p(\overline{x}) \leftarrow & test\_loop\_on(neg\_p(\overline{x})) \bigwedge \\ & test\_only\_negative\_literals\_in\_loop(neg\_p(\overline{x})). \end{array}$$

We illustrate the whole process by means of the following example. It naturally encodes the knowledge that: someone sleeps if he/she is not working (1), someone is working if he/she is not sleeping (3) or if he/she is awake (4) and we know that Susan (6) and Philip (7) are awake.

Example 4.3.2 Consider the following program and the query "$? - \ not(sleep(X))$.".

Program 4.3.2 Program to illustrate the whole process

```
sleep(X) ← not(working(X)).                                    1
                                                               2
working(X) ← not(sleep(X)).                                    3
working(X) ← awake(X).                                         4
                                                               5
awake(susan).                                                  6
awake(philip).                                                 7
```

The evaluation of the query "$? - \ not(sleep(X))$." in the previous program, due to the existence of a loop between $sleep/1$ and $working/1$, does not end or ends with an error under Clark's Semantics or any of its variants. On the contrary, by using our implementation, we obtain the expected results, which are $X = Susan$ and $X = Philip$. We start by computing the dual of the program (in the way we defined in Sec. 4.2) and adding the clauses needed to obtain as results whether the query belong to the Well Founded Model of the program.

Remark. In the program below we perform a transformation that has not been introduced before: the structure "$not(X)$" is changed by "neg_X". This modification is made after the computation of the dual but before adding the clauses needed to get Well Founded Semantics results. The reason for doing it is that $not/1$ is the predicate implementing Negation as Failure and the dual offers us a better procedure to evaluate the negation of a clause p/n, evaluating the clause neg_p/n.

Program 4.3.3 Dual of the program used to illustrate the whole process

```
sleep(X) ← neg_working(X).                                          1
neg_sleep(X) ← working(X).                                          2
                                                                    3
working(X) ← neg_sleep(X).                                          4
working(X) ← awake(X).                                              5
neg_working(X) ← sleep(X).                                          6
neg_working(X) ← neg_awake(X).                                      7
                                                                    8
awake(susan).                                                       9
awake(philip).                                                      10
neg_awake(Y) ← Y ≠ susan, Y ≠ philip.                               11
```

Program 4.3.4 Dual of the program used to illustrate the whole process

```
sleep(X) ← test_no_loops_on(sleep(X)), positive_sleep(X).           1
neg_sleep(X) ← test_no_loops_on(neg_sleep(X)),                      2
               negative_sleep(X).                                   3
neg_sleep(X) ← test_only_negative_literals_in_loop(sleep(X)).       4
                                                                    5
positive_sleep(X) ← neg_working(X).                                 6
negative_sleep(X) ← working(X).                                     7
                                                                    8
working(X) ← test_no_loops_on(working(X)),                          9
             positive_working(X).                                   10
neg_working(X) ← test_no_loops_on(neg_working(X)),                  11
                 negative_working(X).                               12
neg_working(X) ←                                                    13
             test_only_negative_literals_in_loop(working(X)).       14
                                                                    15
positive_working(X) ← neg_sleep(X).                                 16
positive_working(X) ← awake(X).                                     17
negative_working(X) ← sleep(X).                                     18
negative_working(X) ← neg_awake(X).                                 19
                                                                    20
awake(X) ← test_no_loops_on(awake(X)), positive_awake(X).           21
neg_awake(X) ← test_no_loops_on(neg_awake(X)),                      22
               negative_awake(X).                                   23
neg_awake(X) ← test_only_negative_literals_in_loop(awake(X)).       24
                                                                    25
```

```
positive_awake(susan).                                          26
positive_awake(philip).                                         27
negative_awake(Y) ← Y ≠ susan, Y ≠ philip.                      28
```

The SLD-derivation of the resultant program is shown in Fig. 4.3.4. neg_sleep/1 depends on negative_sleep/1, and this one, in turn, on working/1. We have two rules for working/1 and, as the order of the clauses determines the evaluation order, the one that says that working/1 depends on neg_sleep/1 is selected first. Its evaluation fails due to the existence of a negative loop with positive literals involved, so, through backtracking, the second clause is selected. This one says that the evaluation of working/1 depends on awake/1, and its evaluation depends, in turn, on positive_awake/1. The evaluation of positive_awake/1 can be performed by using one of the following rules: "positive_awake(susan)" or "positive_awake(philip)". Both can be used, but again the order of the rules imposes which one is selected first: "positive_awake(susan)". The first solution is found, and $X = susan$ is returned. If a second solution is requested, then through backtracking the second rule is selected, and $X = philip$ is returned.

In [SSW96] the authors present an example of evaluation of SLG resolution. In order to compare the method used there to our method, we expose the program used, the evaluation and result in [SSW96] (Fig. 4.2) and ours.

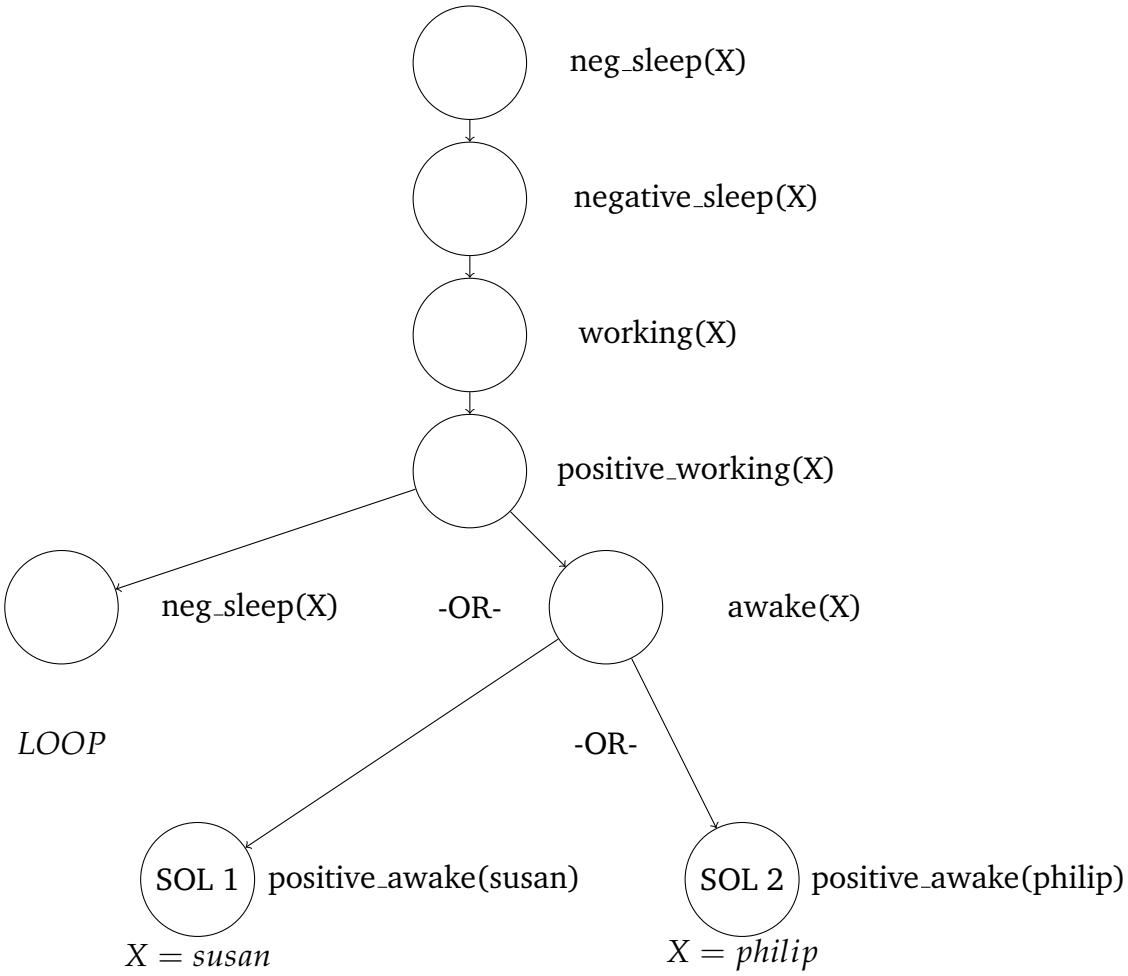

Figure 4.1: SLD-derivation of program 4.3.4

Program 4.3.5 Example program 1.1 from [SSW96]

```
t ← not(p).                                    1
p ← q.                                         2
q ← not(r).                                    3
r ← q, s.                                      4
s ← r.                                         5
```

The query is "? $-$ t.". As can be seen in Fig. 4.2.b, SLG-resolution selects the left-most literal of the first clause, t. The clause for t is marked active, but, because it depends on a negative literal, $not(p)$, it is suspended (marked as "susp") until the evaluation of p is completed. It continues by evaluating the clause for p, which is marked active. As p depends on q, it selects q for evaluation. The clause for q is then activated and, as q depends on $not(r)$, suspended until the evaluation of r is performed. As the evaluation of r depends on the evaluation of q, there is a loop between q and r. When the loop involves one or more negative literals, it is solved by delaying their evaluation. In the example, the evaluation of $not(r)$ is delayed (Fig. 4.2.c). This produces a conditional answer (they are marked as "ans" with a non-empty list of delayed literals) for p and q in the following stage (Fig. 4.2.d). As the evaluation of the clause for r still depends on another literal, s, the clause for s is activated. The loop between r and s is a positive loop (there are no negative literals involved), so both fail (Fig. 4.2.e). The evaluation of $\neg r$ was delayed, but now r has an answer. This answer (failed) succeeds the rule for q, so it is no more delayed. The rule for p succeeds then, and the query "? $-$ t." fails.

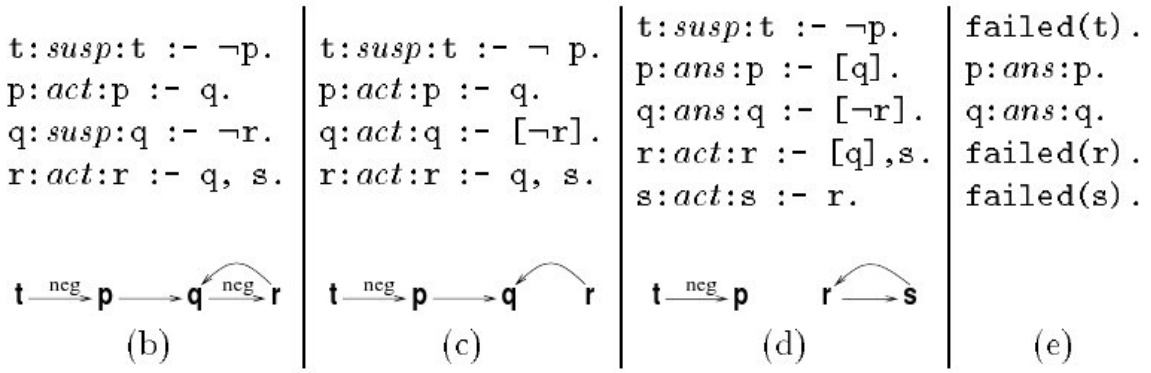

Figure 4.2: Evaluation of query ? $-$ t. in the example 1.1 from [SSW96]

Our method starts by evaluating the dual of the program (program 4.3.6) and doing the modifications needed to obtain results under the Well Founded Semantics (program 4.3.7).

Program 4.3.6 Dual of program 1.1 from [SSW96]

```
t ← neg_p.                          1   neg_q ← r.                      6
neg_t ← p.                          2   r ← q, s.                      7
p ← q.                              3   neg_r ← neg_q ; neg_s.         8
neg_p ← neg_q.                      4   s ← r.                          9
q ← neg_r.                          5   neg_s ← neg_r.                  10
```

Program 4.3.7 Dual of program 1.1 from [SSW96] under WFS.

```
t ← test_no_loops_on(t), positive_t.                                    1
neg_t ← test_no_loops_on(neg_t), negative_t.                            2
neg_t ← test_only_negative_literals_in_loop(t).                         3
positive_t ← neg_p.                                                     4
negative_t ← p.                                                         5
                                                                        6
p ← test_no_loops_on(p), positive_p.                                    7
neg_p ← test_no_loops_on(neg_p), negative_p.                            8
neg_p ← test_only_negative_literals_in_loop(p).                         9
positive_p ← q.                                                         10
negative_p ← neg_q.                                                     11
                                                                        12
q ← test_no_loops_on(q), positive_q.                                    13
neg_q ← test_no_loops_on(neg_q), negative_q.                            14
neg_q ← test_only_negative_literals_in_loop(q).                         15
positive_q ← neg_r.                                                     16
negative_q ← r.                                                         17
                                                                        18
r ← test_no_loops_on(r), positive_r.                                    19
neg_r ← test_no_loops_on(neg_r), negative_r.                            20
neg_r ← test_only_negative_literals_in_loop(r).                         21
positive_r ← q, s.                                                      22
negative_r ← neg_q ; neg_s.                                             23
                                                                        24
s ← test_no_loops_on(s), positive_s.                                    25
neg_s ← test_no_loops_on(neg_s), negative_s.                            26
neg_s ← test_only_negative_literals_in_loop(s).                         27
positive_s ← r.                                                         28
negative_s ← neg_r.                                                     29
```

The SLD-derivation in Fig. 4.3 represents the process performed to evaluate the query "? $-$ t.". We omit in the derivation the rules that take care of loops for the sake of simplicity.

The process starts by selecting the rule for t, and it determines that the rule depends on *neg_p*. In turn, the rule for *neg_p* depends on *neg_q*, and the one for *neg_q* on r. The rule for r depends on the evaluation of the rules for both q and s, and, due

to the left-to-right selection, the one for *q* is selected first. The rule for *q* depends on *neg_r*, and the rule for this one on *neg_q* or *neg_s*. The rule for *neg_q* is not evaluated because our method determines that there is a negative loop on it. This loop involves positive literals, so the derivation fails and, through backtracking, it determines to evaluate the rule for *neg_s*. The rule for *neg_s* depends on *neg_r*, but the rule for *neg_r* is not evaluated because there is a negative loop. As this negative loop has no positive literals involved, it succeeds. The success of *neg_r* makes the rules for *neg_s* and *q* succeed, and the one for *s* is evaluated. The rule for *s* depends on *r*, but it is not evaluated because there is a positive loop. The loop is positive, so *r* fails. As *r* fails, *s*, *neg_q*, *neg_p* and *t* fail. *t* does not belong to the WFM of our program.

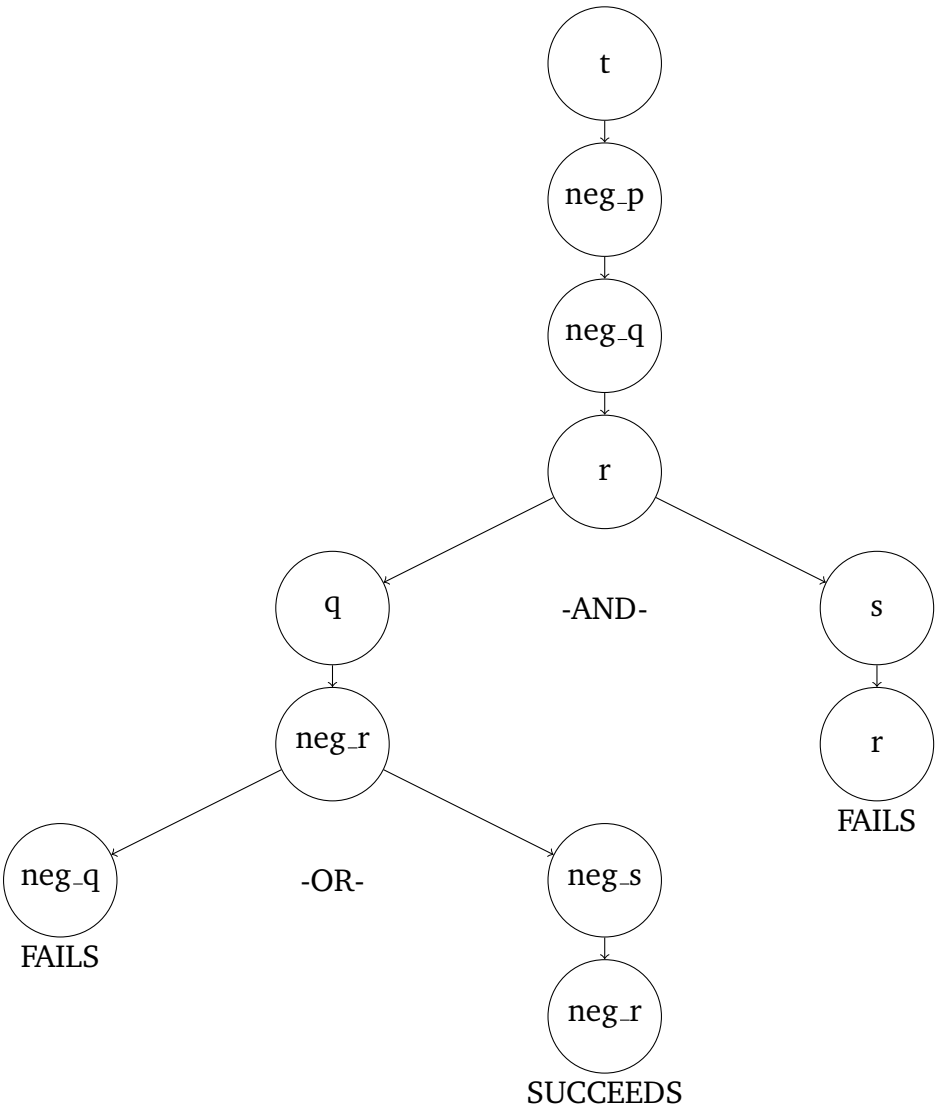

Figure 4.3: SLD-derivation of program 4.3.7 (simplified)

Apart from the fact that our method needs to compute the dual of the program and add the clauses that manage the existence of loops, the main differences against the method in [SSW96], SLG, are the following:

1. the linearity: our method does not delay computations and does not need to notify the delayed literals when an answer is found. This is both an advantage and a disadvantage: by loosing linearity SLG-resolution looses performance, but the programs can be coded in a more declarative way.

2. the storage of the previous returned answers: this is used both to avoid giving the user the same result more than once and to determine, by evaluating a fixed point, that we will not obtain more valid solutions and the computation can be stopped.

4.3.1 Problems of loop detection

Both the differences between our method and that of [SSW96], SLG, pointed out above can in fact be seen as drawbacks of our implementation, and they are both related with the fact that we detect loops solely based on looking at ancestor calls. We illustrate these differences in the examples below.

Example 4.3.3 Suppose the following program and the query "? $- p(X)$.".

Program 4.3.8

```
p(X) ← p(Y).                                                    1
p(a).                                                           2
```

We (informally) demonstrate by induction that $p(X) \in WFM(\Pi)$ by proving that, for some ground substitution σ, $p(X)\sigma$ belongs to $WFM(\Pi)$. As the substitution grounds X, its structure is $\sigma = X/t$ and $t \in \mathcal{H}$ (the term t belongs to the Herbrand Base of the program and the query). So, to demonstrate that $p(X) \in WFM(\Pi)$ we show that it belongs for the base case, where the $\mathcal{H} = \{ a \}$, and for any Herbrand Base having the term a and any other term.

- our Herbrand Base can not be empty because the second clause $p(a)$ has the term a. Moreover, as the term a is coded in our program, the minimal Herbrand Base is $\mathcal{H} = \{ a \}$ and the Herbrand Base of the program always has the term a.

- if our Herbrand Base is $\mathcal{H} = \{ a \}$ then it is clear that $p(a)$ belongs to the WFM of our program. As a is the only value for X and $p(a) \in WFM(\Pi)$, $p(X) \in WFM(\Pi)$.

- if our Herbrand Base is $\mathcal{H} = \{ a, b \}$ then the fixed point is evaluated as follows:

$$I_0 = < \emptyset, \emptyset >$$
$$I_1 = \Omega(I_0) = \Theta_{I_0}^{\uparrow\omega}$$

Since $\Theta_{I_0}^{\uparrow 0} =< \emptyset, \mathcal{H} >$,

$$\Theta_{I_0}^{\uparrow 1} = \Theta_{I_0}(< \emptyset, \mathcal{H} >) = < \{p(a)\},\ \{p(b)\} >$$
$$\Theta_{I_0}^{\uparrow 1} = \Theta_{I_0}(\Theta_{I_0}^{\uparrow 1})$$

Since $\Theta_{I_1}^{\uparrow 0} = < \{p(a)\},\ \{p(b)\} >$,

$$\Theta_{I_1}^{\uparrow 1} = \Theta_{I_1}(\Theta_{I_1}^{\uparrow 0}) = < \{p(a),\ p(b)\}, \{\ \} >$$
$$\Theta_{I_1}^{\uparrow 1} = \Theta_{I_1}(\Theta_{I_1}^{\uparrow 1})$$

and it is easy to see that $\Theta_{I_1}^{\uparrow 2}$ is a fixed point of Θ_{I_1}, i.e.,

$$\Theta_{I_1}^{\uparrow 1} = \Theta_{I_1}^{\uparrow \omega} = I_2$$

As $\{\ a,\ b\ \}$ are the only values for X and $p(a)$ and $p(b)$ belong to $WFM(\Pi)$, $p(X) \in WFM(\Pi)$.

- if our Herbrand Base is $\mathcal{H} = \{\ a,\ b,\ c\ \}$ then the fixed point is evaluated as follows:

$$I_0 =< \emptyset, \emptyset >$$
$$I_1 = \Omega(I_0) = \Theta_{I_0}^{\uparrow \omega}$$

Since $\Theta_{I_0}^{\uparrow 0} =< \emptyset, \mathcal{H} >$,

$$\Theta_{I_0}^{\uparrow 1} = \Theta_{I_0}(< \emptyset, \mathcal{H} >) = < \{p(a)\},\ \{p(b), p(c)\} >$$
$$\Theta_{I_0}^{\uparrow 1} = \Theta_{I_0}(\Theta_{I_0}^{\uparrow 1})$$

Since $\Theta_{I_1}^{\uparrow 0} = < \{p(a)\},\ \{p(b), p(c)\} >$,

$$\Theta_{I_1}^{\uparrow 1} = \Theta_{I_1}(\Theta_{I_1}^{\uparrow 0}) = < \{p(a),\ p(b),\ p(c)\}, \{\ \} >$$
$$\Theta_{I_1}^{\uparrow 1} = \Theta_{I_1}(\Theta_{I_1}^{\uparrow 1})$$

and it is easy to see that $\Theta_{I_1}^{\uparrow 2}$ is a fixed point of Θ_{I_1}, i.e.,

$$\Theta_{I_1}^{\uparrow 1} = \Theta_{I_1}^{\uparrow \omega} = I_2$$

As $\{\ a,\ b,\ c\ \}$ are the only values for X and $p(a)$, $p(b)$ and $p(c)$ belong to $WFM(\Pi)$, $p(X) \in WFM(\Pi)$.

It is clear that no matter which is the Herbrand Universe, $p(X)$ belongs to the Well Founded Model of our program. We apply the method presented here to show its drawbacks.

Remark. We omit the dual of the program for simplicity, and add the WFS behavior only to the original program.

Program 4.3.9

```
p(X) ← test_no_loops_on(p(X)), positive_p(X)                    1
                                                                2
positive_p(X) ← p(Y).                                           3
positive_p(a).                                                  4
```

The SLD-derivation for the evaluation of $p(X)$ is shown in Fig. 4.4. It starts from $p(X)$, which needs $positive\_p(X)$ to be evaluated. The evaluation of $positive\_p(X)$ can be done by using any of the rules for it, $positive\_p(X) \leftarrow p(Y)$ or $positive\_p(a)$, but the way Prolog interpreters choose the rule, when there are multiple options, is

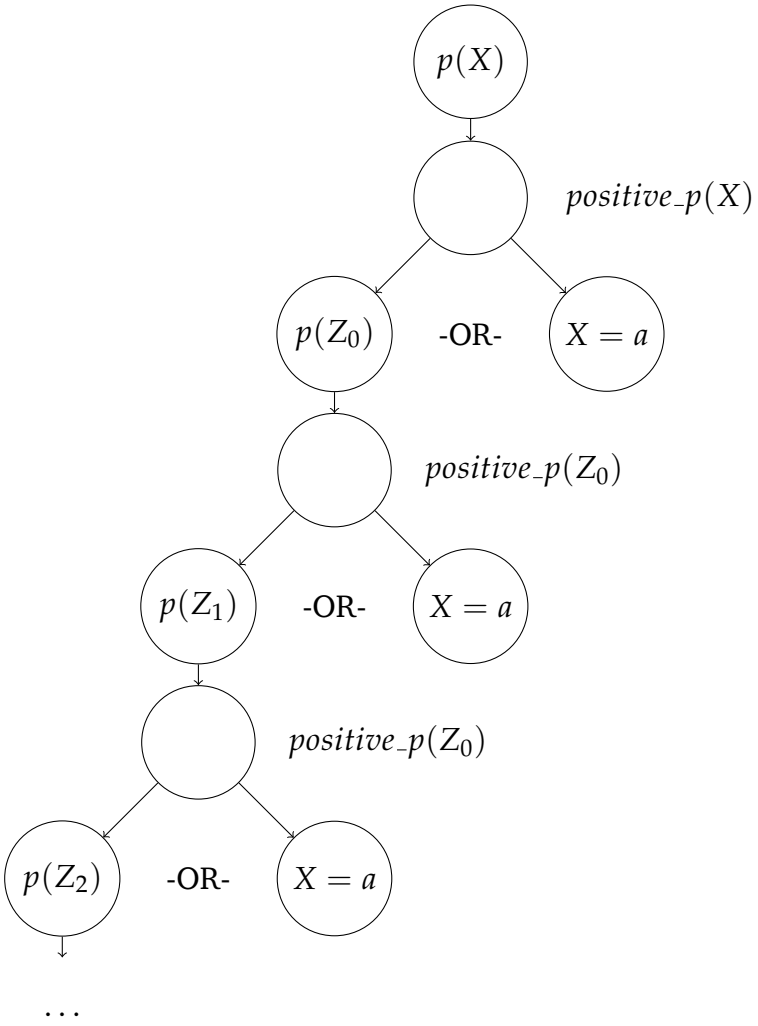

Figure 4.4: SLD-derivation of program 4.3.8

top-down. So, the selected rule for $positive\_p(X)$ is the first one. We rename in the derivation the variable Y by Z_i, to remark that each time we evaluate the rule the variable is new and different from the one used in previous evaluations of the rule. The rule for $p(Z_0)$ determines that there is no loop, that $p(X)$ is different from $p(Z_i)$, and resolves to evaluate $positive\_p(Z_i)$. As before, the Prolog interpreter selects the first rule for $positive\_p(X)$, and loops. The result is that the evaluation never ends and never gives the user any result.

This is the problem of keeping the linearity of the resolution: when in a SLD-derivation the search tree for a given goal has an infinite branch, the order of the clauses can determine if any solutions are given at all. It can be removed by considering $p(Z_{i+1})$ a variant of $p(Z_i)$, and freezing the computation of the rule(s) that loops until the one(s) that does not loop has been evaluated. We show the ideal evaluation in Fig. 4.5, where, after freezing the first rule for $positive\_p(X)$, the first result is found, and each time we evaluate $positive\_p(X)$ we need to freeze its first rule, "$positive\_p(X) \leftarrow p(Y)$", in order to obtain results.

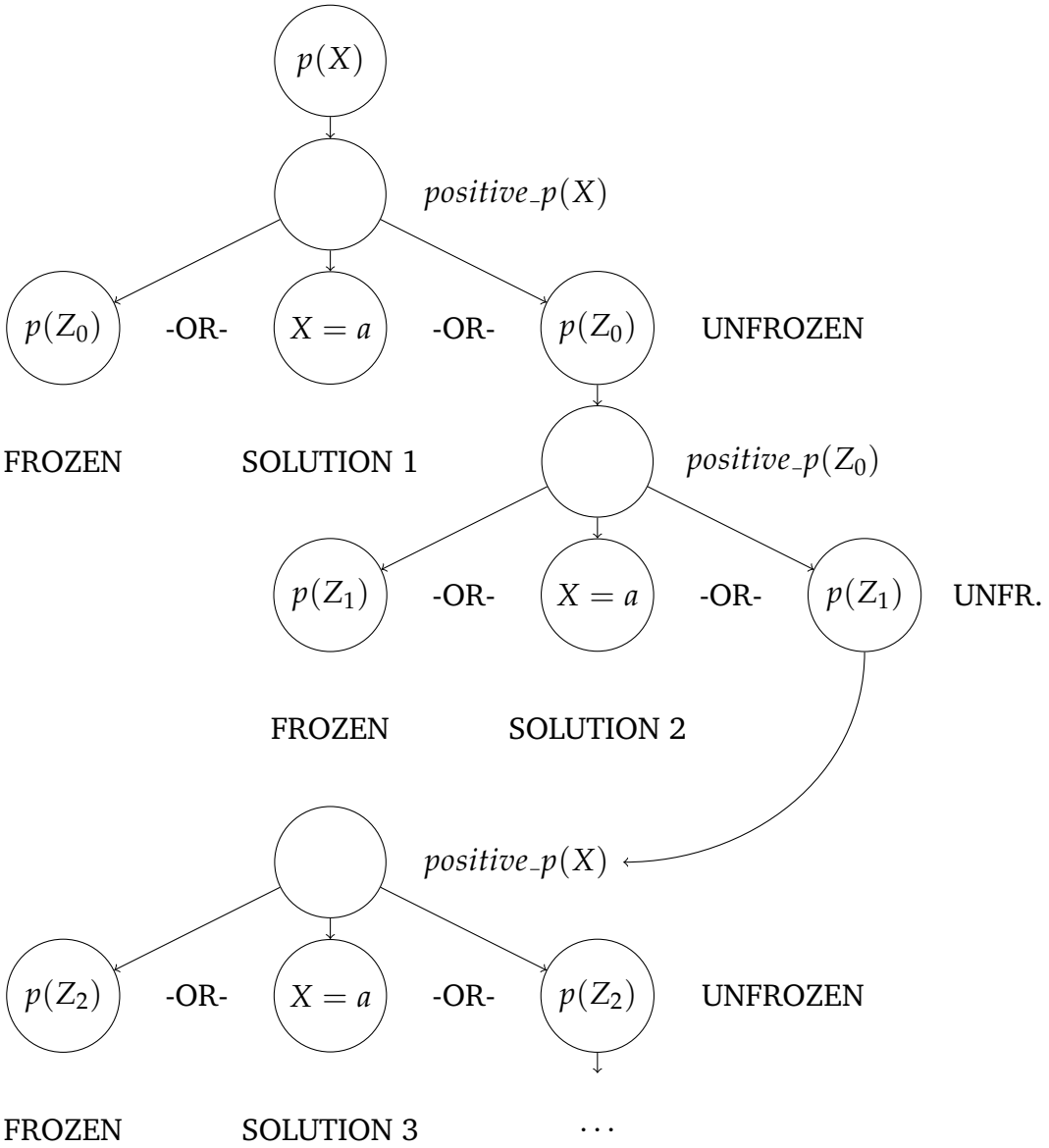

Figure 4.5: Ideal derivation of program 4.3.8

The problem can be removed too by reducing the declarativeness of the programs we code: as the order of the clauses can determine if any solutions are given at all, by coding first the clauses that do not produce an infinite branch in the SLD-derivation tree we avoid the problem. To illustrate this and the second drawback of our implementation, in example 4.3.4 the rules from program 4.3.8 have been reordered so the infinite branch is the rightmost branch of the SLD-derivation tree.

Remark. As program 4.3.10 has the same clauses that program 4.3.8, the Well Founded Model does not need to be evaluated again.

$$WFM(\Pi) = \{\ p(a),\ p(X)\ \}$$

Example 4.3.4 Suppose the following program, obtained by changing the order of the rules in program 4.3.8.

Program 4.3.10

```
p(a).                                                    1
p(X) ← p(Y).                                             2
```

The program obtained from adding the Well Founded Semantics behavior is the following:

Program 4.3.11

```
p(X) ← test_no_loops_on(p(X)), positive_p(X)             1
                                                         2
positive_p(a).                                           3
positive_p(X) ← p(Y).                                    4
```

The SLD-derivation can be seen in Fig. 4.6. To evaluate the query we start as in the previous example, from $p(X)$. It depends again on $positive\_p(X)$, and the evaluation of this one can be done by using the rule $positive\_p(a)$ or the rule $positive\_p(X) \leftarrow p(Y)$. The difference against the SLD-derivation in Fig. 4.4 is that here the rule that is chosen first is "$positive\_p(a)$" instead of "$positive\_p(X) \leftarrow p(Y)$". This makes it possible to obtain solutions, instead of looping while trying to evaluate an infinite branch. Specifically, when evaluating the chosen rule, the first answer is returned, $X = a$.

When asking the system for a second solution, the rule "$positive\_p(X) \leftarrow p(Y)$" is chosen and, as it depends on $p(Y)$ for a new variable Y, the rule for $p(X)$ is evaluated again. The variable Y is again renamed here by Z_i to remark that, each time the rule "$positive\_p(X) \leftarrow p(Y)$" is evaluated, a new different variable is generated. $p(Z_0)$ depends on $positive\_p(X)$, and the evaluation of this one behaves like when looking for the first solution. So, a new solution for $p(X)$ is found. In this case the variable is not bound (X is free), so $p(X) \in WFM(\Pi)$ for any value of X.

The second drawback appears here. When asking for the third solution the expected answer should be "*no*", because there are no more elements in

$WFM(\Pi)$. Instead of that, our method chooses the second rule for $positive\_p(X)$, "$positive\_p(X) \leftarrow p(Y)$". As it depends on $p(Z_1)$, and the evaluation of this one behaves like for the previous results, we obtain a solution: $p(X)$ with X not bound.

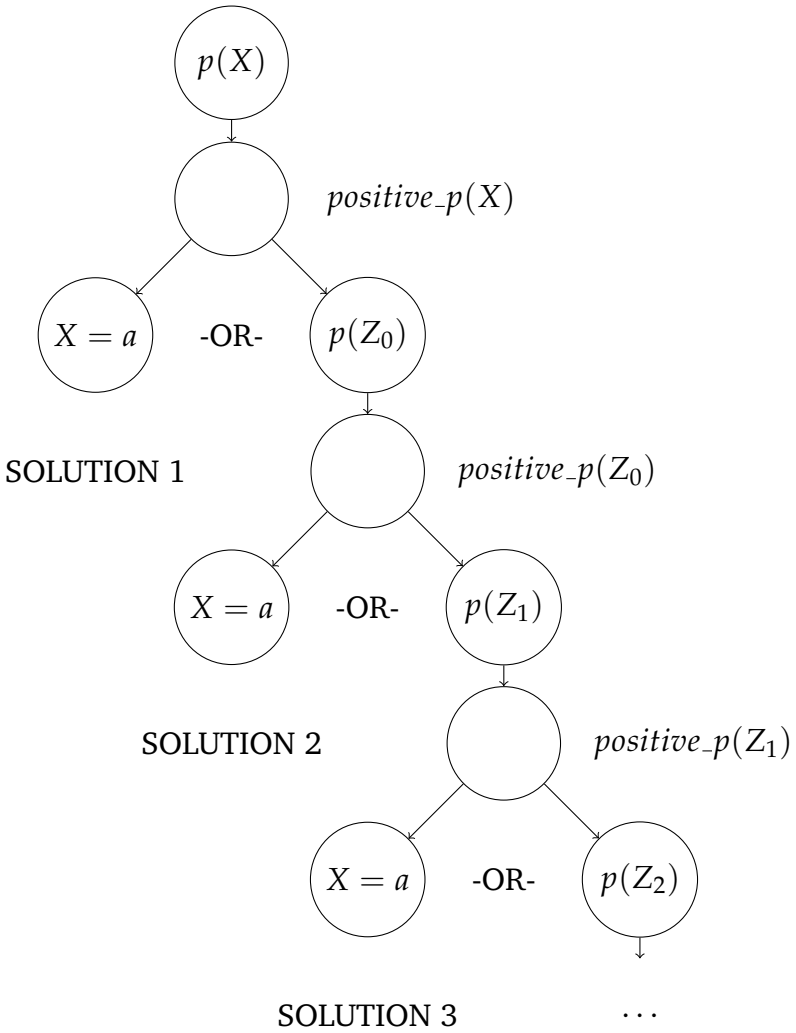

Figure 4.6: SLD-derivation of program 4.3.10.

This third solution has been found before, but Prolog interpreters do not consider it equivalent to the previous one because different computations have been performed to arrive to the solutions. The same happens for the infinite solutions after the third one: all succeed $p(X)$ without binding X.

Getting more than once the same solution can be avoided by saving the previous solutions and checking that the actual has not been returned before, but in this way the evaluation of the infinite branch is still performed and the computation never stops.

The problem is to determine if the infinite branch does not offer any new solution, so we can abort it. The best way is by computing a fixed point to detect if a recursive evaluation does not offer any new solution. The existence of this fixed point will denote the non-necessity to continue the computation, and it can be stopped by failing the derivation branch. In Fig. 4.7 we show a modified version of the derivation in Fig. 4.6. In this one the computation is stopped when it is observed that the evaluation of $p(Z_1)$ needs the evaluation of $p(Z_2)$ (there is a recursive call or loop) and no new solutions have been found in between.

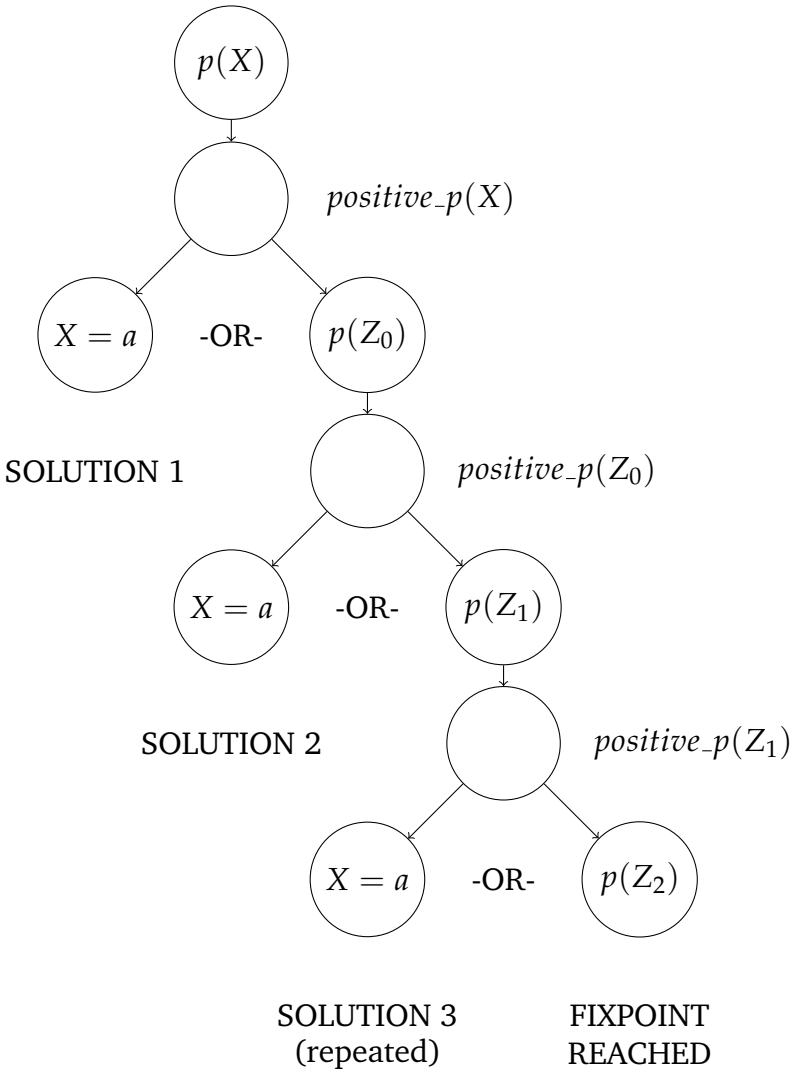

Figure 4.7: Ideal derivation of program 4.3.10.

We illustrate below a method that combines both the freezing and the fixed point features, but none of them is present in the implementation. The reasons for not including them is the necessity to break the SLD-resolution and the loose of performance. Our proposal is that programs be coded in such a way that, if an infinite

branch can exist in the derivation, this one is the rightmost branch of the derivation tree.

The proposed procedure is based on some ideas from [GG01, SW94, RRS+99, CW93, RC94, SyYhY02]. We have divided it in two parts for clarifying: the first avoids the selection of rules that produce infinite branches in the derivation tree when other rules can be selected (the freezing feature), and the second one is in charge of determining when the evaluation of an infinite branch does not offer any new solution (the evaluation of a fixed point). Example 4.3.5 illustrates the whole procedure.

The main idea of the first part is to track the clauses that are evaluated and, when a loop is found (here we consider that evaluating $p(X)$ and $p(Z_0)$ constitutes a loop), the evaluation of the clause that starts the loop ($p(X)$) is canceled. Instead of getting rid of the evaluation of this clause, the branch representing it in the SLD-resolution is moved to the right of the tree. In this way, the execution is reordered and the infinite branches are evaluated only after the finite ones.

The second part determines when the evaluation of a reordered clause does not produce any new solution, and determines to cancel its evaluation. For this purpose a table with the previous solutions is needed, and we need to track whether a predicate has produced new solutions or not.

Example 4.3.5 Suppose the following program:

Program 4.3.12

```
p(X) ← q(X).                                    1
p(a).                                           2
q(X) ← p(Y).                                    3
q(b).                                           4
```

Remark. In this example we do not evaluate the dual of the program or add the clauses that take care of loops. It is done in this way for the sake of clarifying.

Remark. In the derivations shown in this example we write the next literal to be evaluated into the node, and the rule used to determine that this is the next literal to the left of the node. When a loop is found the evaluation of the rule that loops is frozen, and this is denoted by writing below the node the word *FROZEN*. When the evaluation of all the finite rules is performed the frozen rule is unfrozen and evaluated. This is marked to the right of the node.

Fig. 4.8 shows the first step of the derivation, where the loop between $p(X)$ and $p(Z_0)$ is detected and the evaluation of the first clause for $p(X)$, "$p(X) \leftarrow q(X)$.", is frozen until the rules that produce finite branches in the derivation are evaluated.

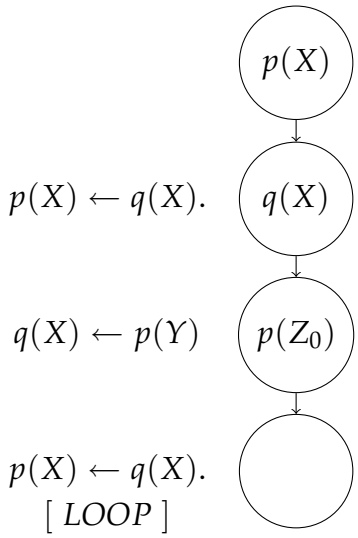

Figure 4.8: Proposed derivation of program 4.3.12 step 1.

Fig. 4.9 shows the second step, where the second available rule for $p(X)$ is selected and evaluated. As it reaches a solution, $X = a$, it is returned.

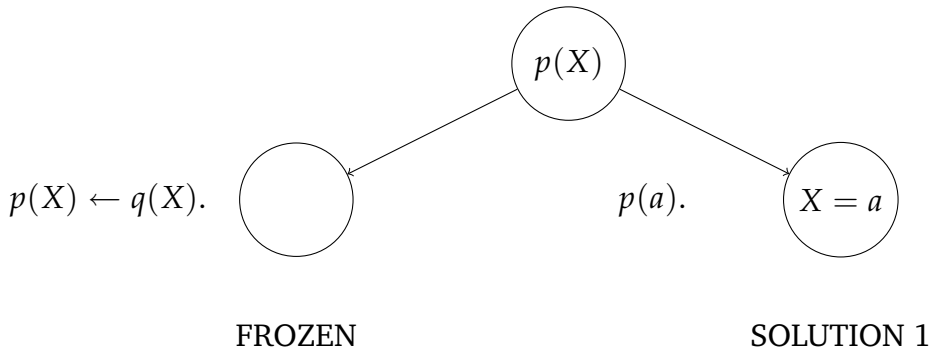

Figure 4.9: Proposed derivation of program 4.3.12 step 2.

Fig. 4.10 shows the third step of the derivation, that is done when a second solution is requested. The frozen clause, "$p(X) \leftarrow q(X)$.", is unfrozen and evaluated. As unfrozen clauses are not considered when checking for loops, a new loop is found on the evaluation of $q(X)$ and the clause "$q(X) \leftarrow p(Y)$." is frozen. We omit the rules of the previous nodes.

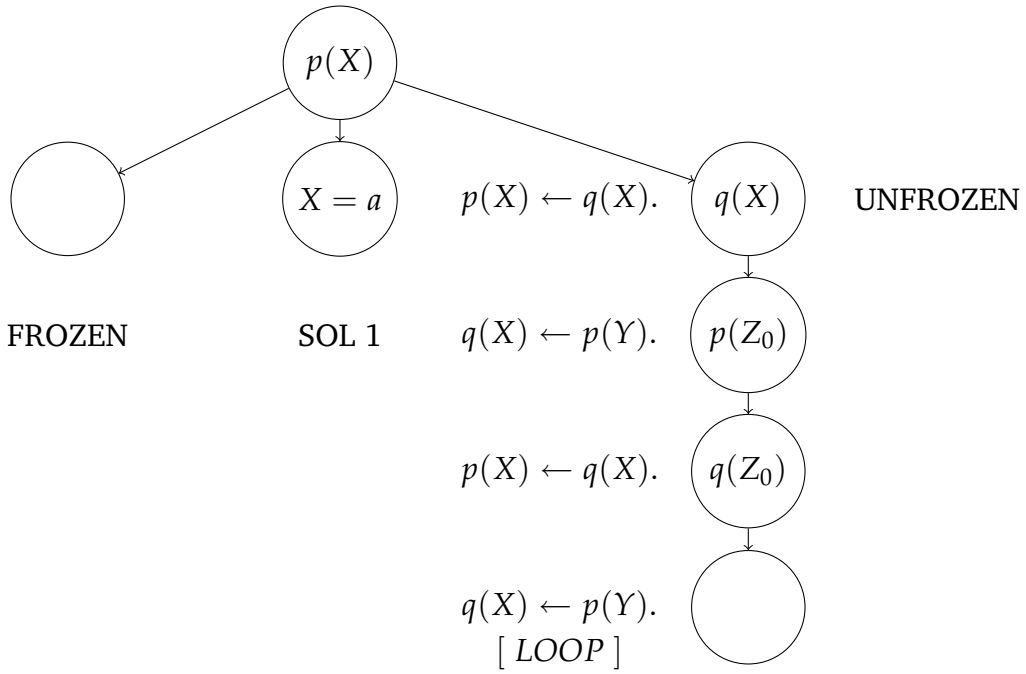

Figure 4.10: Proposed derivation of program 4.3.12 step 3.

In fig. 4.11 the second clause for $q(X)$ is selected and the solution $X = b$ is found.

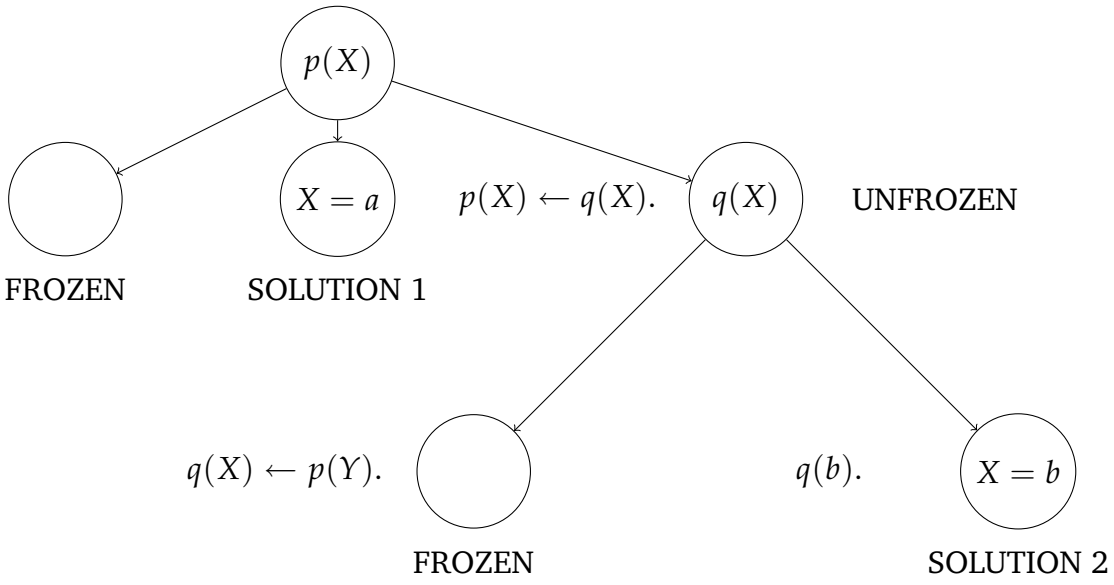

Figure 4.11: Proposed derivation of program 4.3.12 step 4.

Fig. 4.12 shows the fifth step. When looking for the third solution, the frozen clause "$q(X) \leftarrow p(Y)$." is unfrozen and evaluated. During the evaluation a loop on $p(X)$ is found, and the clause "$p(X) \leftarrow q(X)$." is frozen.

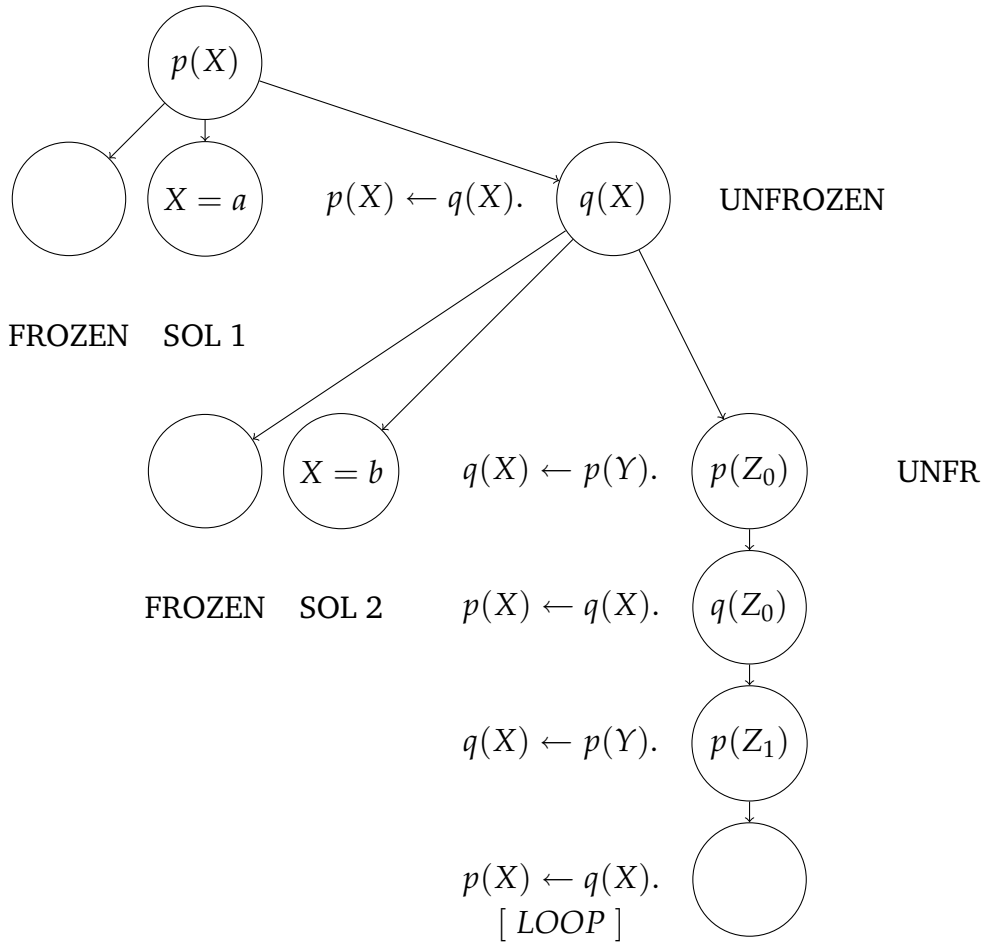

Figure 4.12: Proposed derivation of program 4.3.12 step 5.

In Fig. 4.13 the third solution, $p(X)$ with X unbound, is obtained.

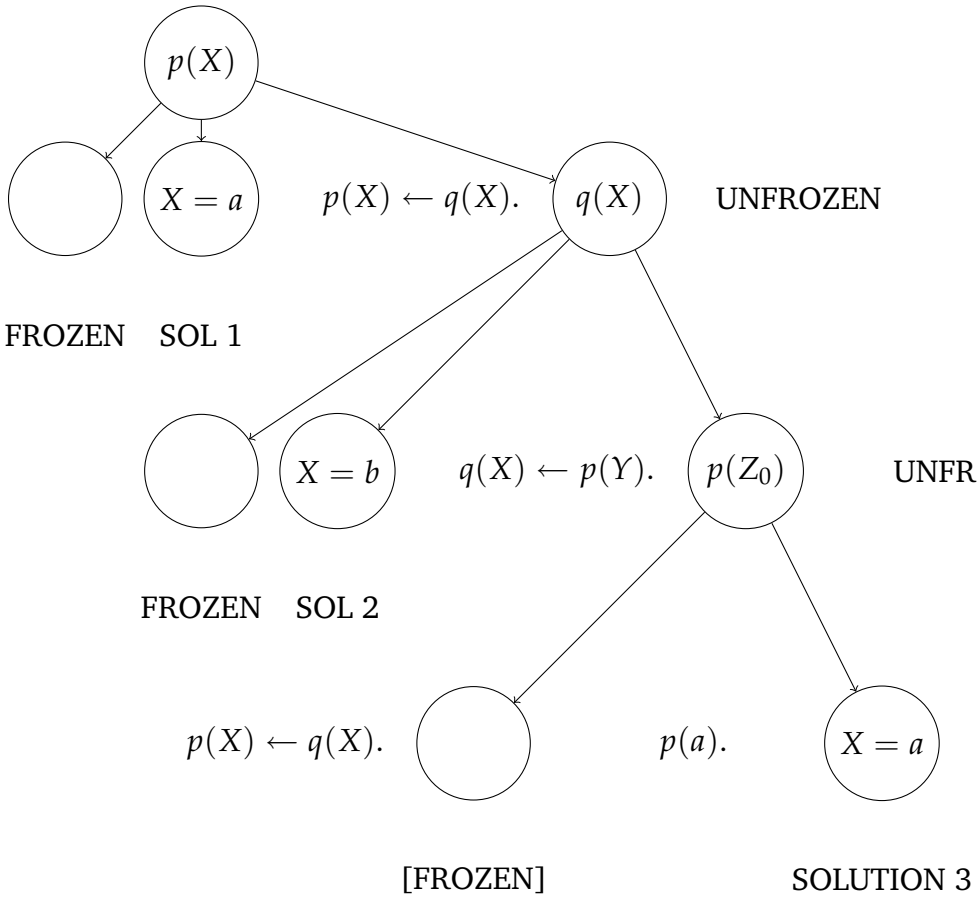

Figure 4.13: Proposed derivation of program 4.3.12 step 6.

When asking for another solution, the method continues evaluating the leftmost branch in the tree, as can be observed in Fig. 4.14. As the unfrozen rules are not considered to determine if there is a loop, a new loop is found on the evaluation of rule "$q(X) \leftarrow p(Y)$".

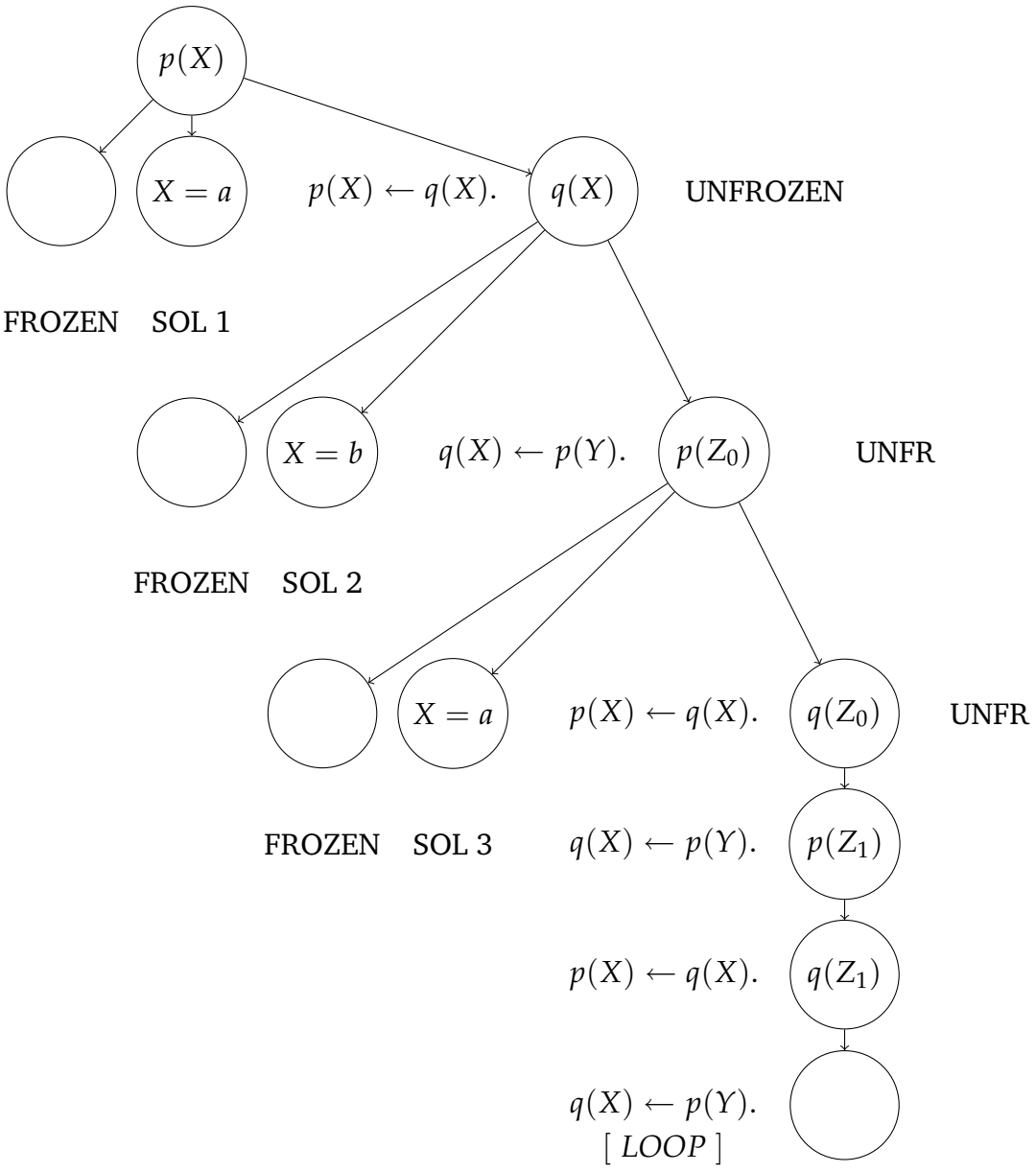

Figure 4.14: Proposed derivation of program 4.3.12 step 7.

The next step of the derivation is split into the figures 4.15 and 4.16. To obtain the fourth solution the method evaluates the finite derivations of $q(Z_0)$ (the rule " $q(X) \leftarrow p(Y)$" has been frozen in the previous step). No new solution is found, because the possible solution, $p(X)$ with X unbound, has been returned as the third solution. So, the frozen clause is unfrozen and reevaluated. As the unfrozen rules are not considered to determine if there is a loop, a new loop is found on the clause "$p(X) \leftarrow q(X)$", and its evaluation is frozen.

Remark. From here onwards we will not repeat the first part of the derivation, because it does not change.

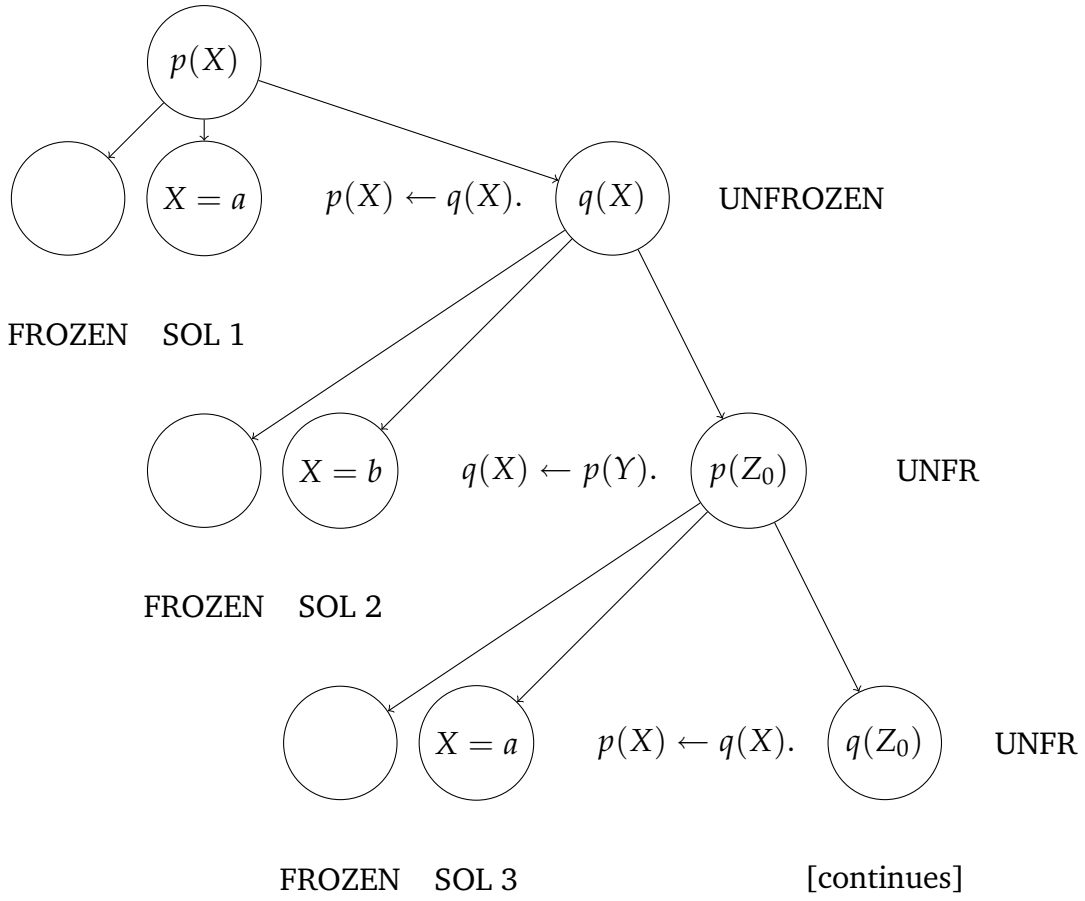

Figure 4.15: Proposed derivation of program 4.3.12 step 8 part 1 of 2.

[continues from Fig. 4.15]

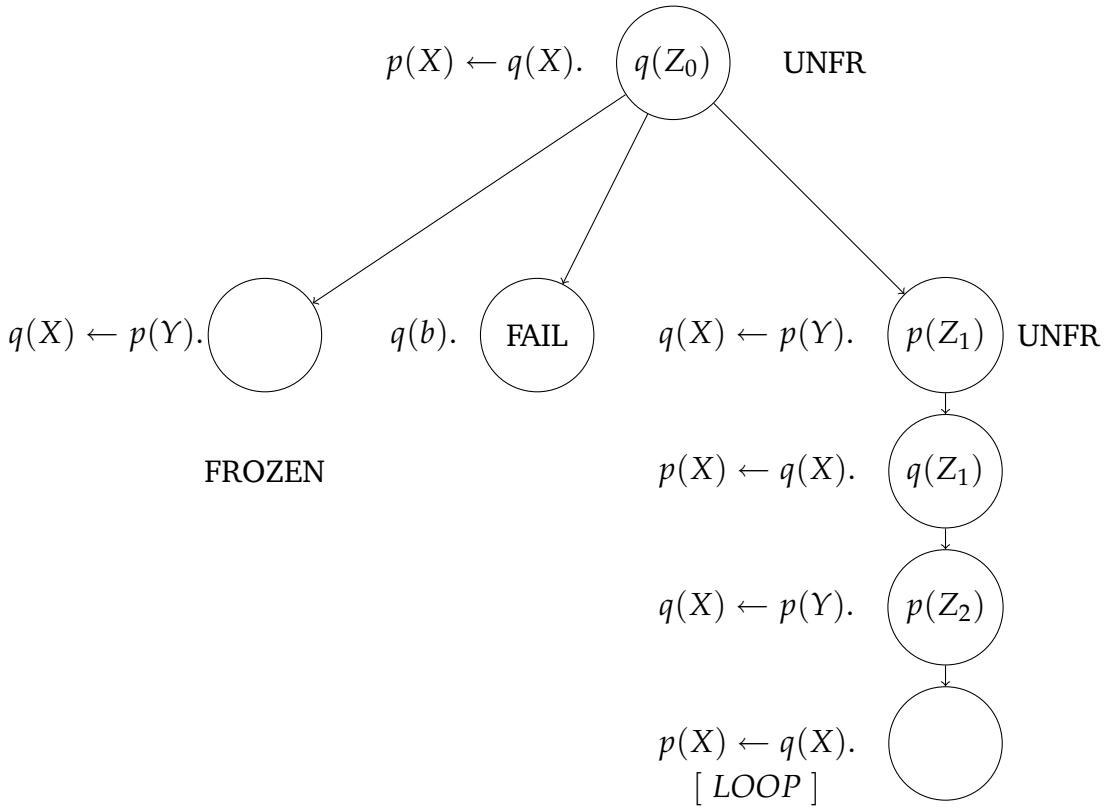

Figure 4.16: Proposed derivation of program 4.3.12 step 8 part 2 of 2.

The evaluation of "$p(X) \leftarrow q(X)$" has been frozen in the previous step, and in Fig. 4.17 the finite solutions for $p(Z_1)$ are evaluated. As no new solutions are found, the frozen rule is unfrozen and reevaluated. Due to the loop on the clause "$q(X) \leftarrow p(Y)$", it is frozen.

[continues from Fig. 4.15]

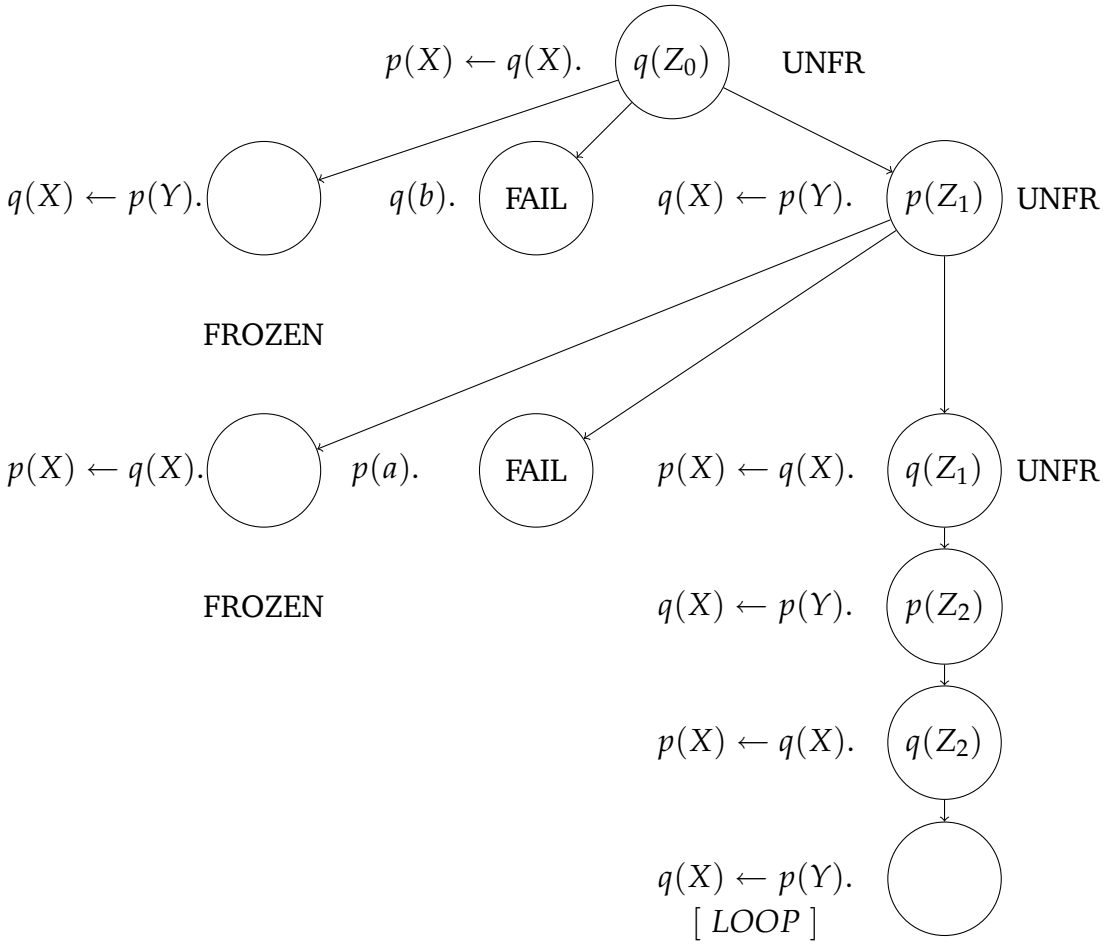

Figure 4.17: Proposed derivation of program 4.3.12 step 9.

As the evaluation of the rule "$q(X) \leftarrow p(Y)$" has been frozen, the finite rules for $q(Z_1)$ are evaluated. After the evaluation of the finite rules (which do not lead us to any new solution), the frozen clause should be unfrozen, but it has been frozen for two times with no new solutions. As it is the second time that its evaluation does not produce any new solution, we can infer that it has reached a fixed point. At this point the evaluation can be (and is) cancelled, because all the solutions have been found.

[continues from Fig. 4.15]

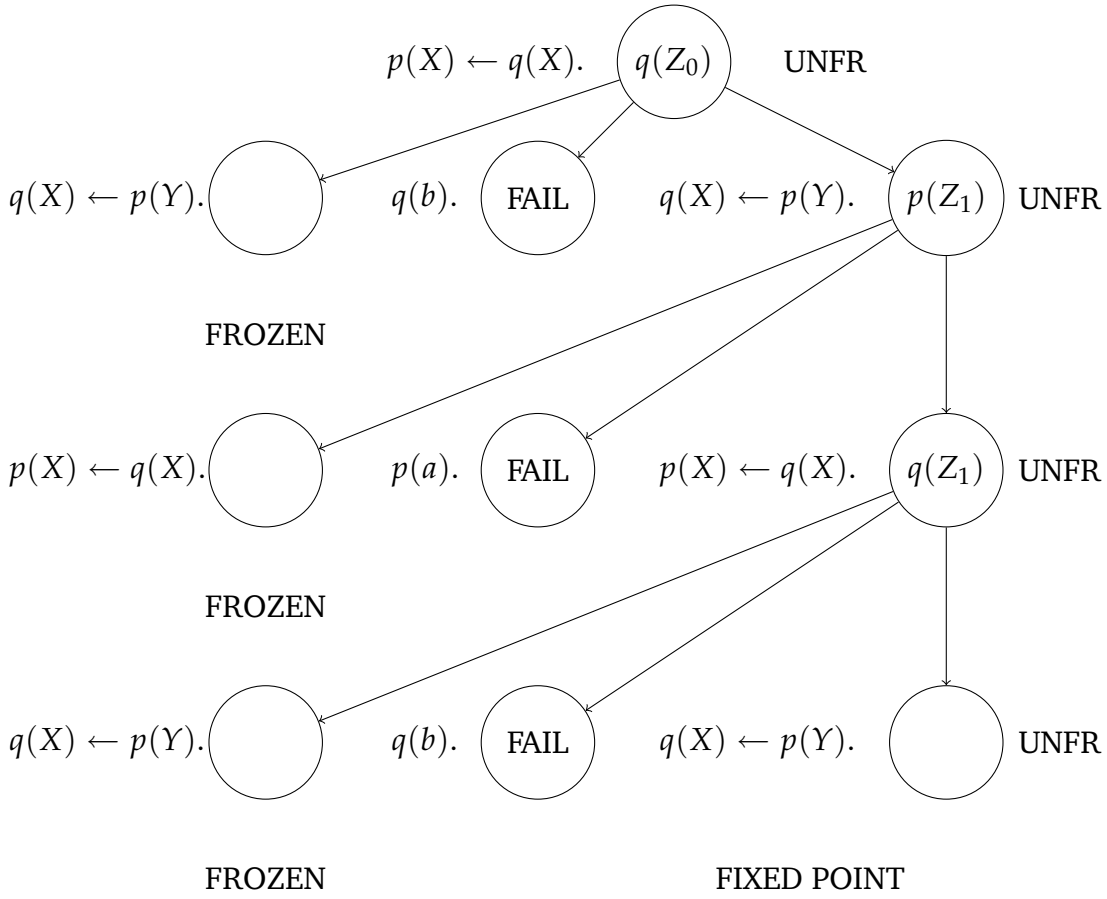

Figure 4.18: Proposed derivation of program 4.3.12 step 10.

4.4 The implementation of inequalities

In section 4.2 we make reference to a predicate, $\neq /2$, that needs further explanation. $\neq /2$ is the predicate in charge of evaluating inequalities. We start by explaining why the Prolog predefined predicate for evaluating inequalities, "$\backslash ==$", is not suitable for evaluating the inequalities that we have in the dual program, and, just after that, we show in detail how $\neq /2$ is implemented.

The Prolog predefined predicate to evaluate inequalities, "$\backslash ==$", can not be used to evaluate the inequalities in the dual program due to two reasons:

- It does not retain information from the disequalities. For example in program 4.4.1 if we evaluate the query $?- p(X).$ the answer will be $X = a$, although we have explicitly coded that X has to be different from a.
  The reason for this behavior is that the information of the disequality is not stored in the variable, and once the disequality code has been evaluated the variable can again take the forbidden value(s).

- As "$\backslash ==$" is not coded in a library but in the Prolog compiler, there is no way to modify the implementation without removing the possibility of using our tool in other Prolog systems. As "$\backslash ==$" is not able to deal with the meta-term $fA(\_)$ (see Sec. 4.2 for more details on the meta-term $fA(\_)$) because it has no representation for it, we can not use the implementation to evaluate disequalities like $s(0) \neq s(fA(\_))$.

Program 4.4.1

```
p(X) ← X \== a, q(X).                                    1
q(a).                                                    2
```

In order to solve these problems we present an implementation of the predicate $\neq /2$ which retains the disequality information by using attributed variables (defined below) and is able to manage disequalities in which one or both terms are composed by using the meta-term $fA(\_)$. Besides, our implementation is able to accumulate information from different inequalities and, whenever it is possible, to simplify the resulting inequality. We start defining attributed variables.

Definition 4.4.1 (Attributed variables)**.** *Attributed variables provide a technique for extending the Prolog unification algorithm [Hol92] by hooking the binding of attributed variables.*

A suspension variable, or an attributed variable, is a variable to which there are suspended agents (or hooks) and attribute values (its name and a value) attached. Agents are registered onto suspension variables by action rules and each attribute is associated to a module where the hook is executed immediately after a successful unification.

The built-in predicates to make use of attributed variables are usually (in some Prolog systems their names change) the following ones:

- *put_attr*/3 *is used to register attribute values,*
- *get_attr*/3 *is used to retrieve attribute values and*
- *del_attr*/2 *is used to remove attribute values.*

The way that agents are registered changes from one Prolog system to another one, but in general they offer some kind of predicate such as $attr\_unify\_hook/3$ *or* $install\_constraint\_portray\_hook/4$*, so the programmer can define the agent behavior.*

Now we present the core (simplified) of our implementation (program 4.4.2), and, after that, we expose some optimizations to increase the speedup of the evaluation of inequalities.

Remark. As the predicate $\neq /2$ can not be coded because there is no symbol for $\neq$, in the code its name is $dist/2$. Besides, $dist/2$ is not defined as an infix operator, but as a prefix operator, so we have $dist(a,b)$ in spite of $a \neq b$.

Program 4.4.2 dist/2 implementation (simplified)

```
dist(T1, T2) ← var(T1), var(T2), T1 == T2, !, fail.                1
dist(T1, T2) ← var(T1), var(T2), add_attribute(T1/T2),             2
                                 add_attribute(T2/T1).             3
dist(T1, T2) ← var(T1), is_forall(T2), !, fail.                    4
dist(T1, T2) ← var(T1), is_functor(T2),                            5
               add_attr(T1/T2).                                    6
dist(T1, T2) ← var(T2), is_forall(T1), !, fail.                    7
dist(T1, T2) ← var(T2), is_functor(T1),                            8
               add_attr(T2/T1).                                    9
                                                                   10
dist(T1, T2) ← is_forall(T1), !, fail.                             11
dist(T1, T2) ← is_forall(T2), !, fail.                             12
                                                                   13
dist(T1, T2) ← functor(T1, Name1, Arity1),                         14
               functor(T2, Name2, Arity2),                         15
               (Name1 \== Name2 ; Arity1 \== Arity2).              16
                                                                   17
dist(T1, T2) ← functor(T1, Name, Arity),                           18
               functor(T2, Name, Arity),                           19
               T1=..[Name|Arguments1],                             20
               T2=..[Name|Arguments2],                             21
               dist_args(Arguments1, Arguments2).                  22
                                                                   23
dist_args([], []) ← !, fail.                                       24
dist_args([A1|L1], [A2|L2]) ←                                      25
               dist(A1, A2) ;                                      26
               dist_args(L1, L2).                                  27
```

The code exposed here is in charge of taking the following decisions:

- accept the inequality because the terms are different at this point and the modifications they can suffer do not affect the decision.
- fail the inequality because the terms unify.
- delay the decision until the variables are grounded by unification.

The last task is performed by adding an attribute to the involved variables, which is done by the predicate *add_attr*/1 Its implementation is presented in program 4.4.3.

Program 4.4.3 add_attribute/1 implementation (simplified)

```
add_attr(T1/T2) ← get_attribute(T1, T1/T3), !,                1
                  remove_attribute(T1),                       2
                  combine_attributes(T2, T3, T4),             3
                  put_attribute(T1, T1/T4).                   4
add_attr(T1/T2) ← put_attribute(T1, T1/T2).                   5
```

The agent to be executed just after the unification of an attributed variable is presented in the program below. Basically it removes the variable's attribute (so the hook is not reevaluated while the agent determines if the inequality holds) and determines if the variable has taken one of the values forbidden by the inequality or not. For doing that we use again the predicate *dist*/2.

Program 4.4.4 verify_attribute/2 implementation (simplified)

```
verify_attribute(Value, T1/T2) ←                              1
                        remove_attribute(T1),                 2
                        dist(Value, T2).                      3
```

Remark. For the sake of simplicity we have not included the code in charge of joining/splitting the list of inequalities that is stored and retrieved from the variable's attribute.

Apart from trivial optimizations, like the elimination of identical inequalities, a huge optimization is carried out by the *dist*/2 predicate: the division of this complex inequalities into a disjunction of simpler and smaller inequalities. We introduce it by means of program 4.4.5.

Program 4.4.5 *bigger_than* **predicate in Peano numbers (exhaustive)**

```
bt(s(X), s(Y)) ← bt(X, Y)                                     1
bt(s(X), 0).                                                  2
bt(V, W) ← ((V, W) ≠ (s(fA(_)), s(fA(_)))),                   3
           ((V, W) ≠ (s(fA(_)), 0)),                          4
           fail.                                              5
```

Taking a look to the first inequality it can be seen that

$$((V,W) \neq (s(fA(\_)),s(fA(\_))))$$

is equivalent to

$$(V \neq s(fA(\_))) \bigvee (W \neq s(fA(\_))).$$

So, instead of having a big and complex inequality, we now have two small and simple inequalities joined by disjunction.

However, this simplification can not always be made: in program 4.4.6 the existing relationship between the first and the second arguments of the predicate *incr*/2 produces an unexpected result. Let's see why:

If we take the program obtained from the negation of the clauses (we omit the code for obtaining WFS results for the sake of clarity), shown in program 4.4.7, and we make the simplification obtained before, we obtain program 4.4.8.

When evaluating the clause, the implementation notices the disjunction and selects the first part of it (and only if it fails will it, through backtracking, look for the second part). Inside the first part, it evaluates the *dist*/2 and adds two attributes: the first to the variable X and the second one to the variable U. As a result, for the clause to succeed it is enough to have the variable U different from the variable X, and, since X is a free variable, this is always the case. $neg\_incr(0,s(0))$ succeeds, which is not the intended meaning. The problem is reproduced in a similar way for the second disjunction of the clause.

Program 4.4.6 *incr* **predicate in Peano numbers**

```
incr(X, s(X)).                                                    1
```

Program 4.4.7 *incr* **predicate in Peano numbers (original+dual)**

```
incr(X, s(X)).                                                    1
neg_incr(U, V) ← (U, V) ≠ (X, s(X)).                              2
```

Program 4.4.8 *incr* **predicate in Peano numbers (erroneous simplification)**

```
incr(X, s(X)).                                                    1
neg_incr(U, V) ← (U ≠ X); (V ≠ s(X)).                             2
```

The reason for this behavior is that we can not split inequalities when there are dependencies between the components of $\overline{y}$ used in $c_i(\overline{x},\overline{y}_i)$. In this case we have to store the whole inequality in each one of the affected variables and test it every time one of the variables is changed by unification.

4.5 *forall*/2 implementation

As introduced in Sec 3.5, the procedure we use to evaluate the universal quantification is based on the construction of tautologies from the answers given by the predicates involved by the universal quantification. The code (simplified) in charge of this task is presented in program 4.5.1.

Program 4.5.1 Core of the *forall*/2 implementation (simplified)

```
intneg_forall(FA_Vars, Preds) ←                                    1
      determine_nfa_vars(Preds, FA_Vars, NFA_Vars), !,             2
      find_sols(FA_Vars, NFA_Vars, Preds, Sols),                   3
      intneg_forall_aux(FA_Vars, NFA_Vars, Sols,                   4
                        Preds, _Expl), !.                          5
```

Basically, $find\_sols(FA\_Vars, NFA\_Vars, Preds, Sols)$ executes each one of the different branches that we have for the predicates *Preds* and stores in *Sols* the solutions for the variables universally quantified (*FA_Vars*), the solutions for the variables that are not universally quantified (*NFA_Vars*) and the predicate(s) from *Preds* used to achieve each one of the solutions.

We have two versions of this predicate. The first one is implemented by *findall* and the second one is implemented by a modified version of *findall* that, instead of retrieving all the solutions in one step, uses Prolog's backtracking to retrieve at each step the previous solutions and a new one. While the first one is more efficient than the second one if the number of solutions is small, it is clear that if we have a predicate with infinite solutions it will not be able to retrieve all of them. On the contrary, if there is a solution, the second one will for sure find it. The algorithm for the modified version of *findall* is shown below.

Algorithm 4.5.1 find_sols implementation

1. Input: $(FA\_Vars,\ NFA\_Vars,\ Preds)$
2. Initialize a variable from which Prolog in backtracking can not remove inserted items. Call it *Solutions*.
3. Execute in Prolog $Call(Preds)$.
   Our new result is the tuple $(FA\_Vars,\ NFA\_Vars,\ Preds)$.
4. Make a new copy of the tuple's variables with *copy_term*, so the variables do not clash with the variables from the following execution.
5. Store in *Solutions* the new tuple.
6. Output: A list with all the existing solutions in *Solutions*.

The predicate $intneg\_forall\_aux(Sols, \_Expl)$ is in charge of the combination of the different solutions to check for tautologies. The algorithm implemented is shown below.

Algorithm 4.5.2 Detection of a tautology, predicate $intneg\_forall\_aux/2$

1. Input: (Sols) [List of solutions with the structure $[(FA\_Vars,\ NFA\_Vars, Preds)]$]
2. Initialize an empty list, *List_Preds* [a].
3. For each variable *Var* in the list of variables *FA_Vars*, loop.
   - 3.1. Take the first solution that still has not been used to free *Var* (the actual variable). Add it to the list *List_Preds*.
   - 3.2. For the solution of the variable,
     - if the variable is free in the solution (it is not bound nor an attributed variable), continue.
     - if this variable is bound, look for a solution in which the variable is an attributed variable and the combination of the actual value and the disequality form a tautology.
       Add the solution used to the list *List_Preds*.
     - if this variable is an attributed variable, the tautology can be built in two ways:
       - by one or more instantiated values from other(s) solution(s).
       - by another solution with an attributed variable in which the disequality forbids different values than the ones forbidden by the actual attributed variable.

       Whatever it is the option taken, add the solutions used to the list *List_Preds*.
4. Test the consistency of the solutions for the variables *NFA_Vars* in the list *List_Preds*. If it is inconsistent, fail. By backtracking look for the next combination of the solutions that frees the variables in *FA_Vars*. If it is consistent, continue.
5. Output: (*List_Preds*) [List of solutions used to freed the variables universally quantified, *FA_Vars*].

[a] *List_Preds* is used to store which solutions has been used to free the list of variables *FA_Vars*.

Remark. The variable $Expl$ in program 4.5.1 (that is not used there) offers the user the possibility to observe which have been the solutions used to free the variables, and this is not only useful for debugging options. In chapter 5 we show the implementation of one extension of our work that makes use of it, abduction.

In order to better illustrate what the implementation of $forall/2$ does, we introduce the following examples. For the sake of simplicity none of them has loops, so, instead of taking the output from the procedure that makes the program obey the Well Founded Semantics (Sec. 4.3), we take the output from the procedure that calculates the dual of the program (Sec. 4.2). As there are no loops, the results are just the same and we can benefit from studying the $forall/2$ behavior in programs that are much simpler.

Example 4.5.1 Suppose that we have the following program:

Program 4.5.2

```
q(Y) ← p(X, Y); neg_ p(X, Y).                                    1
p(0, 1).                                                         2
                                                                 3
neg_q(Y) ← ∀X.(¬ p(X, Y), p(X, Y)).                              4
neg_p(X, Y) ← (X, Y) ≠ (0, 1).                                   5
```

For the query "$?-\neg q(Y)$." the $forall/2$ implementation asks $find\_sols/4$ to obtain all the solutions for X and Y in "$(\neg p(X,Y)\ ,\ p(X,Y))$" and obtains the following results:

- $X \neq 0 \bigwedge X = 0 \bigwedge Y = 1 \rightarrow fail$
- $Y \neq 1 \bigwedge X = 0 \bigwedge Y = 1 \rightarrow fail$

So the universal quantification does not succeed.

Example 4.5.2 Suppose that we have the following program where, compared to the previous one, has the universal quantification applied to a disjunction rather than a conjunction:

Program 4.5.3

```
q(Y) ← p(X, Y), neg_ p(X, Y).                                    1
p(0, 1).                                                         2
                                                                 3
neg_q(Y) ← ∀X.(¬ p(X, Y); p(X, Y)).                              4
neg_p(X, Y) ← (X, Y) ≠ (0, 1).                                   5
```

For the query "$?-\neg q(Y)$." the $forall/2$ implementation asks $find\_sols/4$ to obtain all the solutions for X and Y in "$(\neg p(X,Y)\ ;\ p(X,Y))$" and obtains the following results:

- $X \neq 0$
- $Y \neq 1$
- $X = 0 \bigwedge Y = 1$

After that, $forall/2$ calls $intneg\_forall\_aux/5$, which combines the solutions to obtain tautologies. We have the following possibilities:

- $X \neq 0 \bigvee (X = 0 \bigwedge Y = 1)$
- $Y \neq 1$

Thus, we have two possible solutions for the universal quantification: one which benefits from the fact that X is free and assigns to Y (the variable in "$? - \neg q(Y).$") the value 1 and the other one that joints two solutions to free X and instead of assigning a value to Y adds to the variable an attribute to forbid it to take the value $Y = 1$.

Example 4.5.3 Consider now the program:

Program 4.5.4

```
q(0) ← p(X), r(X).                                  1
p(1).                                               2
r(2).                                               3
                                                    4
neg_q(Y) ← Y ≠ 0.                                   5
neg_q(0) ← ∀X.(¬ p(X); neg_r(X)).                   6
neg_p(X) ← X ≠ 1.                                   7
neg_r(X) ← X ≠ 2.                                   8
```

For the query "$? - \neg q(Y).$" we have two possibilities:

- Y is different from 0 or
- Y is equal to 0, in which case we need to evaluate whether the body of the second rule succeeds to determine if the universal quantification is satisfiable.

In the second possibility the $forall/2$ implementation calls $find\_sols/4$ to obtain all the solutions for X in "$(\neg p(X)\ ;\ r(X))$" and obtains the following results:

- $X \neq 1$
- $X \neq 2$

After that, $forall/2$ evaluates $intneg\_forall\_aux/5$, which obtains only the tautology $X \neq 1 \bigvee X \neq 2$, but this is nevertheless enough for the universal quantification to succeed.

Example 4.5.4 Suppose that we have the following program:

Program 4.5.5

```
q(0) ← p(X).                                                    1
p(1).                                                           2
                                                                3
neg_q(Y) ← Y ≠ 0.                                               4
neg_q(0) ← ∀X.(¬ p(X); neg_r(X)).                               5
neg_p(X) ← X ≠ 1.                                               6
```

For the query "? $-\neg q(Y)$." we have now two possibilities:

- Y is different from 0 or
- Y is equal to 0, in which case we need to evaluate if the body of the second rule succeeds, if the universal quantification succeeds.

In the second possibility $forall/2$ calls $find\_sols/4$ to obtain all the solutions for X in "$(\neg p(X)\ ;\ r(X))$" and obtains the following result:

- $X \neq 1$

After that, $forall/2$ calls $intneg\_forall\_aux/5$. Here it is impossible to build a tautology, and so the universal quantification fails. The only valid solution for the query "? $-\neg q(Y)$." is then $Y \neq 0$.

CHAPTER 5

EXTENSION: ABDUCTION

Although descriptions of abduction date back to the "fourth figure" of Aristotle [Wan94], the philosopher Peirce first introduced the notion of abduction. In [Pei74] he identified three distinguished forms of reasoning:

- **Deduction**, an analytic process based on the application of general rules to particular cases, with the inference of a result.
- **Induction**, synthetic reasoning which infers the rule from the case and the result.
- **Abduction**, another form of synthetic inference, but of the case from a rule and a result.

Peirce further characterized abduction as the "probational adoption of a hypothesis" as explanation for observed facts (results), according to known laws. "It is however a weak kind of inference, because we can not say that we believe in the truth of the explanation, but only that it may be true."

Abduction in logic programming is a method of reasoning in which one chooses the hypothesis that would, if true, best explain the relevant evidence. It starts from a program Π, a set of accepted facts (our query Q), a set of possible hypotheses (the set of abducibles *Abdcs*), a set of integrity rules IR (see Def. 5.1.2 below) and infers their explanations. We will see abductive problems as tuples $< \Pi, Abdcs, IR, Q >$. Let's say that we have an abductive program composed only by the rule:

```
p ← q.
```

$Abdcs = \{ q \}$, the set IR is empty and the query is p. Then it is intended to get an answer $\{ q \}$, meaning that the existence of p in the model of our program can be explained by the fact that q is too in this model.

In contrast with that, logic programming tries to prove that the goal(s) in the query (the accepted facts before, p in the example) are deducible from the hypothesis in the program. Let's say that if we have a logic program composed only by the following rule:

```
p ← q.
```

Then a query $? - p.$ is intended to get an answer *no* because we can not prove p from our program. We need, at least, q to be true in our program for p to succeed, like in the following program:

```
p ← q.                                                    1
q.                                                        2
```

In spite of the fact that implementing abduction by means of logic programs could be a problem (because logic programming proves that the query is deducible from the program, and abductive reasoning looks for the facts that are needed to support this proof), logic-based frameworks are common in the literature [APS04, ST99, TK94, DK02, Men96, CDT91, CP86, Esh93, KM90, Poo89, Poo85, Poo94, Poo93, Pop73, KKT92]. Even Charniak and Shimony [CS94] comment that, pragmatically, a logical framefork for "explanation" is a useful definition since

> ... it ties something we know little about (explanation)
> to something we as a community know quite a bit about (theorem proving)

The only drawback of these implementations is that, as knowledge representation frameworks, they suffer from the problem of the model-theoretic approaches presented in chapters 1 and 2: as originally the existence of variables is not considered or is restricted, the use of variables in abduction suffers from floundering.

Suppose that $Abdcs = \{ parent/2, man/1, woman/1 \}$, IR = $\{ \bot \leftarrow man(X), woman(X) \}$, and the abductive program Π is:

Program 5.0.6

```
father(F,C)←man(F), parent(F,C).                          1
mother(M,C)←woman(M), parent(M,C).                        2
```

Both the queries $father(X,Y)$ and $\neg father(X,Y)$ suffer from floundering: the former when testing that the integrity rule is satisfied and the second one when testing that $father(X,Y)$ can not succeed. Both problems are caused by the use of "negation as failure" to evaluate the negation of the goals: the former tries to evaluate $\neg\bot$ and the second one is a negative query itself.

The expected answers are, for the first query, a list composed with the abductions needed for the predicate $father$ to succeed, $[\ man(X),\ parent(X,Y)\ ]$, and, for the second query, $[\ \neg man(X)\ ;\ \neg parent(X,Y)\ ]$. The meaning of the last one is that if X is not the father of Y is because X is not a man or X is not parent of Y.

The usual method to implement abduction in logic programming is to collect the abductions into a set and test it for consistency, which resembles to two techniques used in chapter 3 for solving the non-ground queries problem. In fact, when 1) collecting all the disequalities and testing them in unification (see Sec. 4.4) and 2) collecting solutions and testing for a tautology (see Sec. 4.5) we are indeed using abductive techniques to solve our problems.

As we arrived to the use of abduction to fix some of our problems, we though that our method for non-ground queries could be used to solve non-ground queries

in abduction, and we were right. We obtained the framework for dealing with non-ground queries that is presented here.

We start by presenting the theoretical bases of abduction (Sec. 5.1) to get to how we translate an abductive problem $< \Pi, Abdcs, IR, Q >$ into a problem that can be solved by any logic programming interpreter (Sec. 5.2). After that we expose how the universal quantification is managed in abduction (Sec. 5.3) and finally how the consistency of the resultant set is tested (Sec. 5.4). The last sections are intended to illustrate the process with some examples (Sec. 5.6) and summarize the work done in abduction (Sec. 5.6).

5.1 The theoretical approach of abduction

As exposed in [Pei74]:

Definition 5.1.1. *Given a set of sentences T (a theory presentation), and a sentence G (observation), to a first approximation, the abductive task can be characterized as the problem of finding a set of sentences Δ (abductive explanation for G) such that:*

1) $T \cup \Delta \models G$

2) $T \cup \Delta$ *is consistent*

This characterization of abduction is independent of the language in which T, G and Δ are formulated. The logical implication $\models$ in 1 can alternatively be replaced by a deduction operator $\vdash$. The consistency requirement 2 is not explicit in Peirce's more informal characterization of abduction, but is a natural further requirement.

In fact, these two conditions, 1 and 2, are too weak to capture Peirce's notion. In particular, additional restrictions on Δ are needed to distinguish abductive explanations from inductive generalizations [CS92]. Moreover, we also need to restrict Δ so that it conveys some reason why the observations hold, e.g. we do not want to explain one effect in terms of another effect, but only in terms of some cause. For both of these reasons, explanations are often restricted to belong to a special pre-specified, domain-specific class of sentences called **abducible**.

Since we are only interested in abduction in logic programs, this definition is too broad. Instead of using it, we use a specialized one, similar to [APS04], that for introducing the notion of (in)consistency makes use of denials.

Definition 5.1.2 (Integrity Rule, modified from [APS04])**.** *An integrity rule for an abductive program Π and a set of abducibles $Abds$ is a denial of the form*

$$\bot \leftarrow L_1, \ldots, L_n$$

where each L_i is a literal.

Definition 5.1.3. *Given an abductive program Π, a set of abducibles $Abds$ and a set of integrity rules IR, Δ is an abductive solution for a query G if and only if*

1. $\Pi \cup IR \cup \Delta \models_{Sem} G$ *and*

2. $\Pi \cup IR \cup \Delta$ *is consistent and*

3. *for every term* A, $A \in \Delta$, *there are a term* B, $B \in Abdcs$, *and a (possible empty) substitution* σ *such that* $A = B\sigma$ *or* $A = (\neg B)\sigma$.

but, as we are restricted to Well Founded Semantics, we can simplify the definition again:

Definition 5.1.4. *Given an abductive program* Π, *a list of abducible predicates* $Abds$, *a set of integrity rules* IR *and a query* Q, Δ *is an abductive solution for our abductive problem* $< \Pi, Abdcs, IR, Q >$ *if and only if*

- *for every term* A, $A \in \Delta$, *there are a term* B, $B \in Abdcs$, *and a (possible empty) substitution* σ *such that* $A = B\sigma$ *or* $A = (\neg B)\sigma$,

- $Q \in WFM\,(\Pi \cup IR \cup \Delta)$ *and*

- $\bot \notin WFM\,(\Pi \cup IR \cup \Delta)$.

Remark. There can be no rule in program Π whose head H is in the set $Abds$ of abducibles or there exists some $A \in Abds$ such that, for some substitution σ, $A\sigma = H$. This does not lead to loss of generality, since any program with abducibles can be rewritten to obey it. For instance, if it is desired to make abducible some objective literal B such that B is the head of a rule, one may introduce a new abducible predicate B', along with a rule " $B : -B'$ ". See e.g. [KKT92].

5.2 Dealing with the abductions

We have defined our abductive problem as a tuple $< \Pi, Abdcs, IR, Q >$, where Π is our abductive program, $Abdcs$ is the set of abducibles, IR the set of integrity rules and Q the query, but we have not defined the syntax of the programs and the query. Although in some papers (like [APS04]) the authors tend to modify the syntax of logic programs to stress the fact that they are abductive programs, our proposal here is to keep the syntax of the abductive programs (and the query) as close as possible to the syntax of logic programs.

By doing that, we can apply the algorithms presented in chapter 4 to obtain the dual of the abductive program and, in the last step, make the needed modifications to the program so that it behaves as an abductive program instead of a logic program. The resultant program will then benefit from the use of the dual program to get the explanations of why a conclusion can not be reached, which are the abductive results for a negative conclusion. So, the syntax used to define abductive programs is the following:

$$abdcs([A_1/Ar_1,\ \ldots,\ A_i/Ar_i]).$$

$$H_1 \leftarrow L_{11},\ ...,\ L_{1n}$$
$$\ldots$$
$$H_m \leftarrow L_{m1},\ ...,\ L_{mn'}$$

$$\bot \leftarrow L_{11},\ ...,\ L_{1k}$$
$$\ldots$$
$$\bot \leftarrow L_{j1},\ ...,\ L_{jk}$$

where "$abdcs([A_1/Ar_1,\ \ldots,\ A_i/Ar_i])$." is the way to declare the set $Abdcs$ of abducibles (A_i/Ar_i is the conventional way in logic programming of referring to the predicate named A_i with arity Ar_i), "$H_m \leftarrow L_{m1},\ ...,\ L_{mn'}$ " is one of our abductive rules and "$\bot \leftarrow L_{j1},\ ...,\ L_{jk}$ " is one of our integrity rules.

Program 5.2.2 shows the result from applying the transformations from chapter 4 to program 5.2.1. For the sake of simplicity from here onwards, unless otherwise stated, we omit the code in charge of getting WFS results.

Remark. As the symbol $\bot$ can not be coded in programs, the term selected to substitute it is $false$, that is assumed to be a new propositional symbol, not appearing elsewhere in the program.

Program 5.2.1

```
abdcs([man/1, woman/1, parent/2]).                          1
                                                            2
father(F,C) ← man(F), parent(F,C).                          3
mother(M,C) ← woman(M), parent(M,C).                        4
                                                            5
false ← man(X), woman(X).                                   6
% X can not be at the same time a man and a woman.         7
```

Program 5.2.2

```
abdcs([man/1, woman/1, parent/2]).                          1
                                                            2
father(F,C)←man(F), parent(F,C).                            3
neg_father(F,C)← neg_man(F) ; neg_parent(F,C).              4
                                                            5
mother(M,C)←woman(M), parent(M,C).                          6
neg_mother(M,C)← neg_woman(M) ; neg_parent(M,C).            7
                                                            8
false ← man(X), woman(X).                                   9
neg_false ← forall(X, ¬man(X); ¬woman(X)).                  10
```

The pending task is to convert the abductive problem, the tuple $< \Pi, Abdcs, IR, Q >$, into a logic problem. It must be done in such a way that, when posing the query Q, we obtain the evidence Δ that can be used to prove the query in the way exposed in Def. 5.1.4, i.e. the abductive solutions for the hypothesis. The conversion is performed by making the following changes:

1. For each clause in our program to have the ability to add the abducibles that explain it to the current set of abducibles, we need two new arguments: One with the previous abducibles' set and another one to return this set plus the new ones that explain its success. For this purpose two new arguments are added in each clause, and when evaluating sub-goals of the clauses these parameters are managed in such a way that the abductions needed by the sub-goals are added to the set of abductions needed by the clause. This is explained in detail just below.

2. For each abducible in the set of abducibles *Abds* we need to build two new clauses, one for abducing its positive version and the other one for abducing its negative version. Besides, when evaluating these clauses and adding the new abducible, we need to test for the consistency of the new set of abducibles. While the first task is simply done by adding the new abduction to the actual list of abductions, the second one is not so simple. It is introduced in detail in Sec. 5.4.

3. The query Q is intended for an abductive program in which the clauses do not have the new arguments to manage the previous set of abductions and the resultant one. In order to fit the new clauses it needs two new arguments: the first that represents the previous set of abductions (and normally will be an empty set) and the second one which is the final result, i.e. the abducibles that explain the query.

4. The integrity restrictions need to be tested when a query is posed to our program. As the integrity rules have been treated as normal rules, the way we test them is by evaluating the goal $\neg false$ ($\neg\bot$), a task that can be done in two places:

   - in the predicate that will be called to evaluate the query,
   - or in the query itself.

   In theory we say that the query is part of our abductive problem, i.e. of the tuple $< \Pi, Abdcs, IR, Q >$. This is done to simplify the theoretical aspects. But, in practice, repeating the whole conversion, if the only thing that changes is the query, is inefficient.

   To enable the possibility to reuse the conversion of the program, the integrity rules and the abducibles, we do the following: instead of testing the integrity restrictions in the predicate that will be called by the query, we add to the query a call to the predicate in charge of testing them. So, the conversion of the query is independent from the conversion of the rest of the problem but, as a drawback, we need to explicitly say in the query that we want to test the integrity rules.

As we have introduced before, in order to return the abducibles that explain one predicate we need to add two new parameters to each clause: *Abds_In* and *Abds_Out*.

- *Abds_Out*: If we want to return the abducibles for a predicate the best way is using a variable, so the programmer can make use of it without too much problems.
- *Abds_In*: A rule might refer to another rule, so we need to obtain the abducibles of the referred rule and add them to the set of abducibles of the referrer rule. The set of abducibles needs to be consistent and, in order to test this consistency, we have two options:
  - Test for consistency before returning the result.
  - Test for consistency each time we add a new abducible.

  Obviously the option selected is the second one, so we do not wait until we have computed a big set of abducibles to say that it is not valid.

  To test consistency of the actual set of abductions we need to know which is this set. *Abds_In* is added for that purpose.

Special care must be taken when adding the variables *Abds_In* and *Abds_Out* to the subgoals of a rule: while when adding them to a disjunction of subgoals we can obtain different explanations for the goal, when adding them to a conjunction we can obtain only one. So, while disjunction $a \leftarrow b \vee c$ is transformed into

$$a(Abds\_In, Abds\_Out) \leftarrow b(Abds\_In, Abds\_Out) \vee c(Abds\_In, Abds\_Out).$$

the conjunction $a \leftarrow b \wedge c$ is transformed into

$$a(Abds\_In, Abds\_Out) \leftarrow b(Abds\_In, Abds\_Aux) \wedge c(Abds\_Aux, Abds\_Out).$$

These transformations are done in a similar way when goals and/or subgoals have arity bigger than zero, as can be seen in program 5.2.3. AI is *Abds_In*, AA is *Abds_Aux* and AO is *Abds_Out*.

Remark. In the last clause of our program it appears a new predicate, $forall/4$, which has not been introduced yet. It is in charge of determining the abductions that explain the satisfaction of the universal quantification, and it is explained in detail in section 5.3.

Program 5.2.3

```
abdcs([man/1, woman/1, parent/2]).                                  1
                                                                    2
father(F, C, AI, AO)← man(F, AI, AA),                              3
                      parent(F, C, AA, AO).                         4
neg_father(F, C, AI, AO)← neg_man(F, AI, AO) ;                      5
                          neg_parent(F, C, AI, AO).                 6
```

```
                                                                    7
mother(M, C, AI, AO)← woman(M, AI, AA),                             8
                      parent(M, C, AA, AO).                         9
neg_mother(M, C, AI, AO)← neg_woman(M, AI, AO) ;                    10
                          neg_parent(M, C, AI, AO).                 11
                                                                    12
man(X, AI, AO) ← add_abdcs([man(X)], AI, AO).                       13
neg_man(X, AI, AO) ← add_abdcs([neg_man(X)], AI, AO).               14
                                                                    15
parent(X, Y, AI, AO) ← add_abdcs([parent(X, Y)], AI, AO).           16
neg_parent(X, Y, AI, AO) ←                                          17
                 add_abdcs([neg_parent(X, Y)], AI, AO).             18
                                                                    19
false(AI, AO) ← man(X, AI, AA), woman(X, AA, AO).                   20
neg_false(AI, AO) ←                                                 21
                  forall(X, ¬man(X); ¬woman(X), AI, AO).            22
```

The conversion and evaluation of the queries $father(F,C)$, $neg\_father(F,C)$, $mother(M,C)$ and $neg\_mother(M,C)$ can be seen below. As the second argument of the predicate in charge of testing the integrity constraints, $neg\_false/2$, does not give us any information, we use the symbol "_" instead of a variable to avoid Prolog to return some value for this variable.

Conversion and evaluation of queries $father(F,C)$, $neg\_father(F,C)$, $mother(M,C)$ and $neg\_mother(M,C)$ in program 5.2.3

```
?- father(F, C, [], AO), neg_false(AO, _).
   AO = [man(F), parent(F,C)] ? ;
   no

?- neg_father(F, C, [], AO), neg_false(AO, _).
   AO = [neg_man(F)] ? ;
   AO = [neg_parent(F, C)] ? ;
   no

?- mother(F, C, [], AO), neg_false(AO, _).
   AO = [woman(F), parent(F,C)] ? ;
   no

?- neg_mother(F, C, [], AO), neg_false(AO, _).
   AO = [neg_woman(F)] ? ;
   AO = [neg_parent(F, C)] ? ;
   no
```

Up to here we have exposed our conversion, but we omitted the explanation of the predicates $forall/4$ and $add\_abdcs/3$. $forall/2$ was the predicate in charge of evaluating the universal quantification in our negation system (see Secs. 3.5 and 4.5), but it was not implemented with the aim of being used for abduction, so it is not ready

to return the abducibles that explain its satisfaction. The problem of the universal quantification in abduction and the needed changes to its implementation are exposed in the following section, and that of consistently adding abductions to a solution is exposed in the section after that.

5.3 Universal Quantification in Abduction

In section 5.2 we have introduced the modifications needed to deal with abduction, but we omitted the management of the universal quantification obtained from the negation of a clause that has a free variable in its body.

For example, when we apply the transformations in chapter 4 to the program 5.3.1 we obtain the program 5.3.2, where the predicate $forall/2$ is the representation of the universal quantification.

Program 5.3.1

```
abdcs([man/1, woman/1, parent/2]).                                  1
                                                                    2
father(F,C)←man(F), parent(F,C).                                    3
mother(M,C)←woman(M), parent(M,C).                                  4
                                                                    5
grandfather(Gf, C) ← father(Gf, P), parent(P, C).                   6
grandmother(Gm, C) ← mother(Gm, P), parent(P, C).                   7
```

Program 5.3.2

```
abdcs([man/1, woman/1, parent/2]).                                  1
                                                                    2
father(F,C)←man(F), parent(F,C).                                    3
neg_father(F,C)← neg_man(F) ; neg_parent(F,C).                      4
                                                                    5
mother(M,C)←woman(M), parent(M,C).                                  6
neg_mother(M,C)← neg_woman(M) ; neg_parent(M,C).                    7
                                                                    8
grandfather(Gf, C) ← father(Gf, P), parent(P, C).                   9
neg_grandfather(Gf, C) ← forall(P, neg_father(Gf, P));              10
                         forall(P, neg_parent(P, C)).               11
                                                                    12
grandmother(Gm, C) ← mother(Gm, P), parent(P, C).                   13
neg_grandmother(Gm, C) ← forall(P, neg_mother(Gm, P));              14
                         forall(P, neg_parent(P, C)).               15
```

If we apply next the conversion of the abducible program into a logic program, we arrive to program 5.3.3, where all predicates (including the $forall/2$) have two new arguments for dealing with the abductions' sets ($forall/2$ is now $forall/4$).

Since, obviously, the universal quantification needs to deal with the sets of abductions, these two new parameters are not only desired but indispensable. The problem is that the implementation presented in Sec. 4.5 is not suitable for dealing with abduction: it is able to determine if the universal quantification is satisfiable, but not which are the abductions that justify this satisfaction. Here we present the modifications done to this implementation to obtain the abductions that guarantee that the argument predicates succeed for all instantiation of its free variables. Consider the program:

Program 5.3.3

```
abdcs([man/1, woman/1, parent/2]).                                    1
                                                                      2
man(X, AI, AO) ← add_abdcs([man(X)], AI, AO).                         3
neg_man(X, AI, AO) ← add_abdcs([neg_man(X)], AI, AO).                 4
                                                                      5
parent(X, Y, AI, AO) ← add_abdcs([parent(X, Y)], AI, AO).             6
neg_parent(X, Y, AI, AO) ←                                            7
                 add_abdcs([neg_parent(X, Y)], AI, AO).               8
                                                                      9
father(F, C, AI, AO)← man(F, AI, AA),                                 10
                      parent(F, C, AA, AO).                           11
neg_father(F, C, AI, AO)← neg_man(F, AI, AO) ;                        12
                           neg_parent(F, C, AI, AO).                  13
mother(M, C, AI, AO)← woman(M, AI, AA),                               14
                      parent(M,C, AA, AO).                            15
neg_mother(M, C, AI, AO)← neg_woman(M, AI, AO) ;                      16
                           neg_parent(M, C, AI, AO).                  17
                                                                      18
grandfather(Gf, C, AI, AO) ← father(Gf, P, AI, AA),                   19
                             parent(P, C, AA, AO).                    20
neg_grandfather(Gf, C, AI, AO) ←                                      21
                   forall(P, neg_father(Gf, P), AI, AO) ;             22
                   forall(P, neg_parent(P, C), AI, AO).               23
                                                                      24
grandmother(Gm, C, AI, AO) ← mother(Gm, P, AI, AA),                   25
                             parent(P, C, AA, AO).                    26
neg_grandmother(Gm, C, AI, AO) ←                                      27
                   forall(P, neg_mother(Gm, P), AI, AO) ;             28
                   forall(P, neg_parent(P, C), AI, AO).               29
```

We start by illustrating the problem of determining which are the abducibles that explain the success of the universal quantification in the previous program. If we make the query

$$? - neg\_grandfather(Gf, C, [\,], AO).$$

to program 5.3.3, as there is a disjunction in the clause, we expect the abductive solution(s) obtained to guarantee the success in the WFM of

1. $forall(P, neg\_father(Gf, P), AI, AO)$ or
2. $forall(P, neg\_parent(P, C), AI, AO)$.

We study first what is needed for the disjunct 1 to succeed and later what is needed by the disjunct 2.

For the disjunct 1 to succeed we need $neg\_father(Gf, P)$ to succeed for all the values of P, which is tested in the same way as in Sec. 4.5: we get the solutions for $neg\_father(Gf, P)$ and try to build a tautology.

To get the solutions for $neg\_father(Gf, P)$ we first need to execute it and, for that, it is mandatory to add the two new arguments needed for abduction. Here, the first one will be an empty list so in the second one we get the abductions needed for this predicate alone to succeed. The solutions obtained are the following:

Evaluation of the translation of query $neg\_father(Gf, P)$ in program 5.3.3

```
?- neg_father(Gf, P, [], AO).
   AO = [neg_man(Gf)] ? ;
   AO = [neg_parent(Gf, P)] ? ;
   no
```

In this example the variable P stays free for the first solution, so we have a tautology. The only task left is to add the abduced solutions of the forall evaluation to the actual set. As the abduced predicate does not have the universally quantified variable P, in this case we do not need to perform any special task. So, to add the new abducibles to the set we can use the predicate $add\_abdcs/3$:

```
add_abdcs(ForAll_Abdcs, AI, AO).
```

that basically (more details on this predicate can be found in Sec.5.4) adds the abductions in the variable $ForAll\_Abdcs$ to the input list of abductions AI, returning the result in its third argument, AO. The evaluation of the second disjunct is rather similar to this one, and we are not introducing it. Instead, we will see how the predicate $forall/4$ behaves when we ask for another solution.

When asking for another solution one could be led to think that the second one will came from the second disjunct, but there is still an usable result in the previous $forall/4$ execution for the first disjunct:

$$AO = [neg\_parent(Gf, P)].$$

So, we take this one and test if P is free or we need to look for another solution to build a tautology. Since it is free, the left task is to add the abductions needed for this solution to succeed to the actual list of abductions, AI. As in this case the universally quantified variable P is used by the abduction, it can not be simply added: we need to remark that in order to succeed the $forall/4$ predicate this abduction has to be in the WFM of Π **for every possible value of the variable** P. For this task a new kind of abducibles is introduced: the abducibles with universally quantified variables.

Definition 5.3.1 (Abducible with universal quantified variable(s))**.** *When to guarantee that an universal quantification succeeds for all instantiations of some predicates' variable(s) we need some abducibles to be in the WFM of the program* Π *for every possible value of the same variable, a more compact representation of this set of abducibles can be used. We can use either of the following syntax for that compact representation, although, when answering queries, Prolog will use only the first one.*

$$forall([Vars],[Abducibles])$$
$$or$$
$$\forall_{Vars}\,[Abducibles]$$

The intended meaning is that we need in the WFM of the program Π *the abducibles Abducibles for every possible value of the variables in Vars to guarantee that an universal quantification succeeds.*

The results obtained when executing the query "$? - \ neg\_grandfather(Gf, C, [\,], AO)$." are presented below, where the first two solutions belong to the disjunct 1 and the third one to the disjunct 2.

Evaluation of query $neg\_grandfather(Gf, C, [\,], AO)$ **in program 5.3.3**

```
?- neg_grandfather(Gf, C, [], AO).
   AO = [neg_man(Gf)] ? ;
   AO = [forall(P, [neg_parent(Gf, P)])] ? ;
   AO = [forall(P, [neg_parent(P, C)])] ? ;
   no
```

Since the previous example does not build tautologies (the universally quantified variable is always free), we introduce the following example to illustrate the process performed when a tautology needs to be built to make the variable free.

Program 5.3.4

```
abdcs([r/1]).                                   1
                                                2
p ← q(X).                                       3
                                                4
q(1) ← r(1).                                    5
q(2) ← r(2).                                    6
```

For the sake of simplicity we first calculate the dual by using the process exposed in Sec. 4.2.

Program 5.3.5

```
abdcs([r/1]).

p ← q(X).
neg_p ← forall(X, neg_q(X)).

q(1) ← r(1).
q(2) ← r(2).
neg_q(X) ← X ≠ 1, X ≠ 2.
neg_q(1) ← neg_r(1).
neg_q(2) ← neg_r(2).
```

After that we add the WFS behavior by using the procedure exposed in Sec. 4.3.

Program 5.3.6

```
abdcs([r/1]).

p ← test_no_loops_on(p), positive_p.
neg_p ← test_no_loops_on(neg_p), negative_p.
neg_p ← test_loop_on(neg_p),
        test_only_negative_literals_in_loop(neg_p).

positive_p ← q(X).
negative_p ← forall(X, neg_q(X)).

q(X) ← test_no_loops_on(q(X)), positive_q(X).
neg_q(X) ← test_no_loops_on(neg_q(X)), negative_q(X).
neg_q(X) ← test_loop_on(neg_q(X)),
        test_only_negative_literals_in_loop(neg_q(X)).

positive_q(1) ← r(1).
positive_q(2) ← r(2).
negative_q(X) ← X ≠ 1, X ≠ 2.
negative_q(1) ← neg_r(1).
negative_q(2) ← neg_r(2).
```

Finally we add the arguments and clauses needed to get the abductions that explain the hypotheses.

Program 5.3.7

```
abdcs([r/1]).

p(AI, AO) ← test_no_loops_on(p(AI, AO)),
            positive_p(AI, AO).
neg_p(AI, AO) ← test_no_loops_on(neg_p(AI, AO)),
```

```
                    negative_p(AI, AO).                                    6
  neg_p(AI, AO) ← test_loop_on(neg_p(AI, AO)),                             7
       test_only_negative_literals_in_loop(neg_p(AI, AO)).                 8
                                                                           9
  positive_p(AI, AO) ← q(X, AI, AO).                                       10
  negative_p(AI, AO) ← forall(X, neg_q(X), AI, AO).                        11
                                                                           12
  q(X, AI, AO) ← test_no_loops_on(q(X), AI, AO),                           13
                  positive_q(X, AI, AO).                                   14
  neg_q(X) ← test_no_loops_on(neg_q(X), AI, AO),                           15
             negative_q(X, AI, AO).                                        16
  neg_q(X) ← test_loop_on(neg_q(X), AI, AO),                               17
      test_only_negative_literals_in_loop(neg_q(X), AI, AO).               18
                                                                           19
  positive_q(1, AI, AO) ← r(1, AI, AO).                                    20
  positive_q(2, AI, AO) ← r(2, AI, AO).                                    21
  negative_q(X, A, A) ← X ≠ 1, X ≠ 2.                                      22
  negative_q(1, AI, AO) ← neg_r(1, AI, AO).                                23
  negative_q(2, AI, AO) ← neg_r(2, AI, AO).                                24
                                                                           25
  r(X, AI, AO) ← add_abdcs(r(X), AI, AO).                                  26
  neg_r(X, AI, AO) ← add_abdcs(neg_r(X), AI, AO).                          27
```

If we try to obtain the abducibles that explain the existence of neg_p in the WFM of Π, the result for the query "$? - neg\_p([\,], AO)$.", the program will determine that it needs $forall(X,\ neg\_q(X), AI, AO)$ to succeed, and the solutions for $neg\_q(X, AI, AO)$ are the following:

Evaluation of query $neg\_q(X, AI, AO)$ in program 5.3.7

```
?- neg_q(X, [], AO).
   X ≠ 1 ∧ X ≠ 2
   AO = [] ? ;
   X = 1
   AO = [neg_r(1)] ;
   X = 2
   AO = [neg_r(2)] ;
   no
```

So, for the predicate $forall/4$ to succeed we have to build a tautology, which in this case is

$$(\ X \neq 1 \wedge X \neq 2\ ) \vee X = 1 \vee X = 2.$$

As when building the tautology we just append the abductions needed by each result to the list AI that initially is empty, finally we obtain the list

$$[\ neg\_r(1),\ neg\_r(2)\ ]$$

which does not contain the variable X. So, the final result does not need to quantify the abductions universally:

Evaluation of query $neg\_p([\,], AO)$. **in program 5.3.7**

```
?- neg_p([], AO).
   AO = [neg_r(1), neg_r(2)] ? ;
   no
```

As a final remark in this section, we show a subset of the universally quantified abductions that we have introduced before: the abductions with constrained universally quantified variables. Consider the following final program:

Program 5.3.8

```
abdcs([g/1]).                                                          1
                                                                       2
g(X, AI, AO) ← add_abdcs(g(X), AI, AO).                                3
neg_g(X, AI, AO) ← add_abdcs(neg_g(X), AI, AO).                        4
                                                                       5
test(AI, AO) ← neg_p(X, AI, AA), q((X, AA, AO).                        6
neg_test(AI, AO) ← forall(X, (p(X); neg_q(X)), AI, AO).                7
                                                                       8
p(X, AI, AO) ← q(X, AI, AA), g(X, AA, AO).                             9
neg_p(X, AI, AO) ← q(X, AI, AO) ; g(X, AI, AO).                        10
                                                                       11
q(X, AI, AO) ← neg_memberchk(X, [1,2,3], AI, AO).                      12
neg_q(X, AI, AO) ← memberchk(X, [1,2,3], AI, AO).                      13
                                                                       14
memberchk(X, [X|L]).                                                   15
memberchk(X, [Y|L]) ← memberchk(X, L).                                 16
                                                                       17
neg_memberchk(X, Y) ← (X, Y) ≠ (fA(_), [fA(_)|fA(_)]).                 18
neg_memberchk(X, [Y|L]) ← (X, [Y|L]) ≠ (Z, [Z,fA(_)]),                 19
                          neg_memberchk(X, L).                         20
```

and suppose that we want to obtain the abductions that explain *neg_test*, the results for $neg\_test([\,], AO)$. When evaluating $forall(X,\ (p(X);\ neg\_q(X)),\ AI,\ AO)$ we need to obtain the solutions for $p(X);\ neg\_q(X)$, which are:

Evaluation of query $p(X);\ neg\_q(X)$ **in program 5.3.8**

```
?- p(X, [], AO) ; neg_q(X, [], AO).
   X ≠ 1 ∧ X ≠ 2 ∧ X ≠ 3
   AO = g(X) ? ;
   X = 1
   AO = [] ? ;
   X = 2
   AO = [] ? ;
   X = 3
   AO = [] ? ;
   no
```

As can be seen, from the list of solutions we can build the tautology $(X \neq 1 \wedge X \neq 2 \wedge X \neq 3) \vee X = 1 \vee X = 2 \vee X = 3$, so we only have to join the abductions obtained for each of the solutions used to build the tautology. When joining, the result is:

$$g(X)\ [X \neq 1 \wedge X \neq 2 \wedge X \neq 3]\ ;$$

and after that we determine that, as the variable X is the same variable we had in $forall$, we need to quantify it universally. The result that $forall/4$ returns is

$$forall(X, g(X))[X \neq 1 \wedge X \neq 2 \wedge X \neq 3]$$

which we resolved to call constrained universally quantified variable.

Definition 5.3.2 (Abducible with constrained universal quantified variable(s))**.** *When to guarantee that an universal quantification succeeds for all instantiations of some predicates' variable(s) we need some abducibles to be in the WFM of the program Π for every possible value of the same variable, it can happen that the variable has been constrained before.*

In this case the constraints are transferred to the universal quantification, because abducing a predicate for a variable's value that the variable is not allowed to take is nonsense.

As in the universal quantified variables, a more compact representation of this set of abducibles can be used. Its syntax is the following:

$$forall([Vars], [Abducibles])\ \ [Constraints]$$

For the sake of clarifying we may also use one of the following representations:

$$forall([Var1_{Constraints}, \ldots, Varn_{Constraints}],\ [Abducibles])$$
$$or$$
$$\{\ \forall_{[Var1,\ldots,Varn]}\ [Abducibles]\ \}\ \ [Constraints]$$

The meaning of this abduction is that we need to assume the predicate for all the values of the variable but the ones in the constraint.

Remark. Although we try to make the syntax more legible by putting the constraints as close as possible to the variables, by writing

$$\begin{array}{ll} \{\ p(X_{X\neq a})\} & \text{instead of } \ \{\ p(X)\ \}\ [X \neq a] \\ & \text{or} \\ \{\ \forall X_{X\neq a}.\ p(X)\ \} & \text{instead of } \ \{\ \forall X.\ p(X)\ \}\ [X \neq a], \end{array}$$

in some cases it is preferable to write the constraints out of the set, like in

$$\{\ p(X),\ neg\_p(Y)\}\ [X \neq Y] \quad \text{instead of} \quad \{\ p(X_{X\neq Y}),\ neg\_p(Y_{Y\neq X})\},$$

where the notation can lead to think that there are two constraints while there is only one. Both syntaxes are equivalent.

5.4 Testing the consistency of the abductions' set

In the previous sections 5.3 and 5.2 we always referred to the consistency of the abduction set as a task that has to be done when adding a new abducible to the actual set, i.e. as a task of the predicate

$$add\_abdcs(New\_Abducibles, AI, AO).$$

Now we define how the consistency of a set of abductions is evaluated each time we add a new abduction. We start by defining what is considered inconsistency:

Definition 5.4.1 (Inconsistency of a set of abductions)**.** *A set of abductions is inconsistent if and only if for at least one ground atom A, both A and its negation $\neg A$ belong to the set.*

Since the set is in general not ground, having instead predicates with variables (possibly universally quantified and/or with restrictions) that stand for a set of ground literals (resulting from this grounding), in order to determine if a set of abducibles is consistent we need to look first at the different forms of literals that we can have as abductions.

F1) A literal (or its negation) with its variables instantiated. For example, $p(a)$ (or $neg\_p(a)$).

F2) A literal with one or more of its variables free. For example, $p(X)$ (or $neg\_p(X)$) or $p(X,a)$ (or $neg\_p(X,a)$).

F3) A literal with one or more of its variables constrained. For example, $p(X_{X\neq a})$ (or $neg\_p(X_{X\neq a})$) or $p(X_{X\neq a},a)$ (or $neg\_p(X_{X\neq a},a)$).

F4) A literal with one or more of its variables universally quantified. For example, $\forall X.\ p(X)$ (or $\forall X.\ neg\_p(X)$) or $\forall X.\ p(X,a)$ (or $\forall X.\ neg\_p(X,a)$).

F5) A literal with one or more of its variables universally quantified and constrained. For example, $\forall X_{X\neq a}.\ p(X)$ (or $\forall X_{X\neq a}.\ neg\_p(X)$) or $\forall X_{X\neq a}.p(\ X,a)$ (or $\forall X_{X\neq a}.\ neg\_p(X,a)$).

We examine the combinations to determine which of them make the set inconsistent, i.e. the combinations such that its grounding contains an inconsistency. We restrict the combinations to the cases in which one abducible is positive and the other one is negative, since it is clear that if both are positive or negative abducibles the combination can not lead to inconsistency.

C1) If we have a ground literal (F1) and its negation (F1), for example $p(a)$ and $neg\_p(a)$, it is clear from Def. 5.4.1 that the set is inconsistent.

C2) If for a literal we have a ground instance (F1) and its non-ground negation (F2), for example $neg\_p(a)$ and $p(X)$, the set can be made consistent since the variable can take a value different from the one taken by the ground instance (we need to constraint the variable). So, if in the example we constraint the variable X to $X \neq a$ the set $\{neg\_p(a),\ p(X_{X\neq a})\}$ is consistent.

C3) If for a literal we have a ground instance (F1) and its non-ground negation constrained (F3),

- if the variable can not take the ground value then the set is consistent without modifying the abductions, for example $neg\_p(a)$ and $p(X_{X\neq a})$.
- if the variable can take the ground value, the variable must be constrained so it can not take the ground value. For example, in $neg\_p(a)$ and $p(X)_{X\neq b}$ the resultant set is $\{\ neg\_p(a),\ p(X_{X\neq b,X\neq a})\ \}$.

C4) If for a literal we have a ground instance (F1) and its non-ground negation universally quantified and not constrained (F4), for example $p(a)$ and $forall(X,\ neg\_p(X))$, the set is inconsistent. The universal quantification forces us to abduce the literal for all the possible values of the variable, and one of these values is taken by the ground instance of the literal, so the combination form an inconsistent set. In the example, $neg\_p(a)$ is one of the ground instances included in the abduction $forall(X,\ neg\_p(X))$, and the combination $\{p(a),\ neg\_p(a)\}$ is inconsistent.

C5) If for a literal we have a ground instance (F1) and its non-ground negation universally quantified and constrained (F5),

- if the universal quantification can not take the ground value then the combination is consistent. For example, $p(a)$ and $forall(X_{X\neq a}, neg\_p(X))$.
- if the universal quantification can take the ground value then the set is inconsistent. The reason is the same as in C4. For example, $p(a)$ and $forall(X_{X\neq b}, neg\_p(X))$: $neg\_p(a)$ is included into $forall(X_{X\neq b}, neg\_p(X))$ and the set formed by $p(a)$ and $neg\_p(a)$ is inconsistent.

C6) If for a literal we have its non-ground positive and negative versions (F2), they can be added since a constraint assures that they can not take the same ground values. For example, $p(X)$ and $neg\_p(Y)$ can form the set $\{\ p(X),\ neg\_p(Y)\}\ [X \neq Y]$.

C7) If for a literal we have its non-ground version (F2) and its non-ground negated version constrained (F3), they can be added since a constraint assures that they can not take the same ground values. For example, $neg\_p(Y)$ and $p(X_{X\neq a})$ can form the set $\{\ neg\_p(Y),\ p(X)\}\ [X \neq a,\ X \neq Y]$.

C8) If for a literal we have its non-ground version (F2) and its non-ground negated version universally quantified (F4), they can not form a consistent set. The universally quantified one takes all the existing values and the other one can not be bound to any value without making the set inconsistent. For example, $neg\_p(Y)$ and $forall(X,\ p(X))$.

C9) If for a literal we have its non-ground version (F2) and its non-ground negated version universally quantified and constrained (F5), they can form a consistent set by constraining the values that the first one can take to the values

that the second one can not take. For example, the union of $neg\_p(Y)$ and $forall(X_{X \neq a},\ p(X))$ form the set $\{\ neg\_p(Y_{Y=a}),\ forall(X_{X \neq a},\ p(X))\ \}$ or, simplified, the set $\{\ neg\_p(a),\ forall(X_{X \neq a},\ p(X))\ \}$.

C10) If for a literal we have its non-ground constrained version (F3) and its negated non-ground constrained version (F3) they can form a consistent set by constraining the variables not to take the same value. For example, $neg\_p(Y_{Y \neq a})$ and $p(X_{X \neq b})$ can form the set $\{\ neg\_p(Y),\ p(X)\ \}\ [Y \neq a, X \neq b, Y \neq X]$.

C11) If for a literal we have its non-ground constrained version (F3) and its non-ground negated version universally quantified (F4), they can not form a consistent set. The reason is the same as in C8. For example, $neg\_p(Y_{Y \neq a})$ and $forall(X,\ p(X))$.

C12) If for a literal we have its non-ground constrained version (F3) and its non-ground negated version universally quantified and constrained (F5), then

- if the first one can be constrained to take the values that the second one can not take then they can form a consistent set. For example, $neg\_p(Y_{Y \neq a})$ and $forall(X_{X \neq c},\ p(X))$ can form the consistent set $\{\ neg\_p(Y_{Y \neq a,\ Y = c}),\ forall(X_{X \neq c},\ p(X))\ \}$ or, simplified, $\{\ neg\_p(c),\ forall(X_{X \neq c},\ p(X))\ \}$.
- if the first one can not be constrained to take the values that the second one can not take, then the set is inconsistent. For example, $neg\_p(Y_{Y \neq c})$ and $forall(X_{X \neq c},\ p(X))$.

C13) If for a literal we have its non-ground negated version universally quantified (F4) and its non-ground negated version universally quantified (F4) then the consistent set can not be formed. Both of them need all the ground instances of the literal for the variable, and this means that for every instance of the literal the set will be inconsistent. For example, $forall(X,\ p(X))$ and $forall(Y,\ neg\_p(Y))$: since they need all the instances, we can take a ground value a and see that the combination of $p(a)$ and $neg\_p(a)$ is inconsistent.

C14) If for a literal we have its non-ground version universally quantified (F4) and its non-ground negated version universally quantified and constrained (F5) then the consistent set can not be formed. We can always find a valid ground value for the constrained one and see that this value is in the set of values that the other one needs to take too. For example, $forall(X,\ p(X))$ and $forall(X_{X \neq a},\ neg\_p(X))$: we can take a ground value b and see that the combination of $p(b)$ and $neg\_p(b)$ is inconsistent.

C15) If for a literal we have its positive and negative non-ground versions universally quantified and constrained (F5) then we can not form a consistent set. The reason is similar to the one in C14: we can find a valid ground value for both of them and see that this value is in the set of values that the other one needs to take too. For example, $forall(X_{X \neq a},\ p(X))$ and $forall(X_{X \neq b},\ neg\_p(X))$: we can take a ground value c and see that the combination of $p(c)$ and $neg\_p(c)$ is inconsistent.

As introduced before, we use an inductive method to assure the consistency of the set:

- the empty set is obviously consistent.
- each time a new abduction is added to the set it is tested that the combination of this new abduction with the existing ones does not make the set inconsistent.

Since testing the combination by using rules C1 to C15 is clearly inefficient, we reformulate the rules in the following ones. The difference is that the new ones only determines when adding a new abducible to the set makes it inconsistent:

R1) The set becomes inconsistent if we add to it a literal (for example, $p(X)$) when we have in the set its negation universally quantified (for example, $\forall Y.neg\_p(Y)$).

This applies too if we want to add the literal with the variable grounded (for example, $p(b)$).

One exception to this rule occurs when the universally quantified formula is constrained (for example, $\forall Y_{Y \neq a}.neg\_p(Y)$) and the variable (or constant) of the predicate to add (X in the example) can be constrained to the values that the universally quantified variable can not take (a in the example). In this case the variable has to be constrained to take only those values (in the example X has to be constrained to take the value a).

R2) The set becomes inconsistent if we add to it a literal universally quantified (for example, $\forall Y.\ p(Y)$) when we have in the set its negation (for example, $p(X)$).

This applies too if we have in the set the literal with the variable grounded (for example, $p(b)$).

The exception of the rule R1 applies here too.

R3) The set becomes inconsistent if we add to it a literal (for example, $p(X)$ or $neg\_p(X)$) and we have in the set its negation (respectively, $neg\_p(X)$ or $p(X)$).

This applies too if we have in the set the literal with the variable grounded (for example, $p(b)$ or $neg\_p(b)$) or if we have to add the literal with the variable grounded (for example, $p(b)$ or $neg\_p(b)$).

One exception to this rule occurs in the following cases:

- when the literals are ground and do not take the same value for the variable (for example, $p(b)$ and $neg\_p(c)$).
- when one of them is not ground ($neg\_p(X)$ or $p(X)$) and the other one is. In this case the variables of the first one must be constrained so they can not take the values taken by the first one (if the first one is $p(d)$ or $neg\_p(d)$, respectively $neg\_p(X_{X \neq d})$ or $p(X_{X \neq d})$).
- when both of them are not ground ($neg\_p(X)$ or $p(X)$ and $p(X)$ or $neg\_p(X)$) a constraint must be added to the variables to assure that they do not take the same values (the resultant set for both cases is $\{\ neg\_p(X_1),\ p(X_2)\ \}\ [X_1 \neq X_2]$).

We finally illustrate how this behaves by means of some examples:

Example 5.4.1 Suppose that we have a set of abductions $AO = [neg\_r(X)]$ and we have to add the abduction $r(Y)$. As the second one is the negation of the first one, it can not be added unless we constraint them to take different values. So, the result is: $AO = [neg\_r(X), r(Y)][X \neq Y]$.

Example 5.4.2 Suppose that we have a set of abductions $AI = [neg\_r(X),\ r(Y)]\ [X \neq Y]$ and we have to add the abduction $r(1)$. As the new abduction is included in $r(Y)$, we do not need to explicitly add it, but we need to constraint $neg\_r(X)$ so X can not be bound to 1. So, the result is $AO = [neg\_r(X),\ r(Y)]\ [X \neq Y,\ X \neq 1]$.

Example 5.4.3 Suppose that we have a set of abductions $AI = [neg\_r(X),\ r(Y)]\ [X \neq Y,\ X \neq 1]$ and we have to add the abduction $neg\_r(1)$. As there is a constraint over X that does not allow us to take this value, the set is inconsistent and we fail to add the new abducible.

Example 5.4.4 Suppose that we have a set of abductions $AI = [neg\_r(X),\ r(Y)]\ [X \neq Y,\ X \neq 1]$ and we have to add the abduction $forall(Z,\ r(Z))$. As there is a negated version of $r(Z)$ in our set, the result is inconsistent and we fail to add the new abduction.

Example 5.4.5 Suppose that we have a set of abductions $AI = [neg\_r(X),\ r(Y)]\ [X \neq Y,\ X \neq 1]$ and we have to add the abduction $forall(Z_{Z \neq 2},\ r(Z))$. As $r(Z)$ needs to be abduced for every value but $Z = 2$, we can constraint $neg\_r(X)$ to $X = 2$. Besides, as Y is constrained to have a value different from the one taken by X, and $X = 2$, then $Y \neq 2$. So, $r(Y)$ is contained into $forall(Z_{Z \neq 2},\ r(Z))$ and the result from adding the new abduction is $AO = [neg\_r(2),\ forall(Z_{Z \neq 2},\ r(Z))]$.

5.5 Illustrative examples

In this section we expose the conversion of some abductive problems into logic programs, and we test the results obtained against the definition of abductive solution in Def. 5.1.4.

Example 5.5.1 The abductive problem is formed by the following abductive program Π (program 5.5.1), that declares $Abds = \{\ p/1\ \}$ and $IR = \{\ \}$, and the query q.

Program 5.5.1

```
abdcs([p/1]).                                    1
                                                 2
q ← p(X).                                        3
```

The corresponding logic program is

Program 5.5.2 Conversion of the abductive program Π into a logic program

```
q(AI, AO) ← test_no_loops_on(q(AI, AO)),                          1
            positive_q(AI, AO).                                   2
                                                                  3
positive_q(AI, AO) ← p(X).                                        4
                                                                  5
neg_q(AI, AO) ← test_no_loops_on(neg_q(AI, AO)),                  6
                negative_q(AI, AO).                               7
neg_q(AI, AO) ←                                                   8
      test_only_negative_literals_in_loop(neg_q(AI, AO)).         9
                                                                  10
negative_q(AI, AO) ← forall(X, neg_p(X), AI, AO).                 11
                                                                  12
p(X, AI, AO) ← add_abdcs([p(X)], AI, AO).                         13
neg_p(X, AI, AO) ← add_abdcs([neg_p(X)], AI, AO).                 14
```

The result from converting the query q is $q([\,], AO)$, and its evaluation is:

Evaluation of query $q([\,], AO)$ in program 5.5.2

```
?- q([], AO).
   AO = [p(X)] ? ;
   no
```

In order to determine that the result obtained is correct, according to Def. 5.1.4, we have to test that

- for every term A, $A \in \Delta$, there are a term B, $B \in Abdcs$, and a (possible empty) substitution σ such that $A = B\sigma$ or $A = (\neg B)\sigma$. $p(X)$ is the only term in Δ and $p(X) \in Abds$ so, for σ the empty substitution, $A = B\sigma$.
- $Q \in WFM\,(\Pi \cup IR \cup \Delta)$. As the meaning of $p(X)$ is that we need one ground instance of the predicate $p/1$, we add $p(a)$. The result is program 5.5.3, where it can be seen that Q belongs to the WFM by simply making the query q.
- $\bot \notin WFM\,(\Pi \cup IR \cup \Delta)$. As $IR = \{\,\}$, it does not need to be checked.

Program 5.5.3

```
q ← test_no_loops_on(q), positive_q.                              1
neg_q ← test_no_loops_on(neg_q), negative_q.                      2
neg_q ← test_only_negative_literals_in_loop(neg_q).               3
                                                                  4
p(X) ← test_no_loops_on(p(X)), positive_p(X).                     5
neg_p(X) ← test_no_loops_on(neg_p(X)), negative_p(X).             6
neg_p(X) ←                                                        7
```

```
      test_only_negative_literals_in_loop(neg_p(X)).          8
                                                              9
positive_q ← p(X).                                            10
negative_q ← forall(X, neg_p(X)).                             11
                                                              12
positive_p(a).                                                13
negative_p(X) ← X ≠ a.                                        14
```

Evaluation of query q in program 5.5.3

```
?- q.
   yes
?-
```

Example 5.5.2 The abductive problem is formed by the previous abductive program Π (program 5.5.1), that declares $Abds = \{ p/1 \}$ and $IR = \{ \}$, and the query *neg_q*. As explained before, we do not need to convert again the whole problem, it suffices to convert the query Q.
The result from converting the query *neg_q* is $neg\_q([\,], AO)$, and its evaluation is:

Evaluation of query $neg\_q([\,], AO)$ in program 5.5.2

```
?- neg_q([], AO).
   AO = [forall(X, neg_p(X))] ? ;
   no
```

In order to determine that the result obtained is correct, according to Def. 5.1.4, we have to test that

- for every term A, $A \in \Delta$, there are a term B, $B \in Abdcs$, and a (possible empty) substitution σ such that $A = B\sigma$ or $A = (\neg B)\sigma$. $neg\_p(X)$ is the only term in Δ and $p(X) \in Abds$ so, for σ the empty substitution, $A = \neg B\sigma$.
- $Q \in WFM\,(\Pi \cup IR \cup \Delta)$. As the meaning of $forall(X, neg\_p(X))$ is that we need all the ground negative instances of the predicate $p/1$, we add $neg\_p(X)$ (which is going to succeed for all of them). The result is program 5.5.4, where it can be seen that Q belongs to the WFM by simply making the query *neg_q*.
- $\bot \notin WFM\,(\Pi \cup IR \cup \Delta)$. As $IR = \{ \}$, it does not need to be checked.

Program 5.5.4

```
q ← test_no_loops_on(q), positive_q.                          1
neg_q ← test_no_loops_on(neg_q), negative_q.                  2
neg_q ← test_only_negative_literals_in_loop(neg_q).           3
                                                              4
p(X) ← test_no_loops_on(p(X)), positive_p(X).                 5
neg_p(X) ← test_no_loops_on(neg_p(X)), negative_p(X).         6
```

```
neg_p(X) ←                                                          7
      test_only_negative_literals_in_loop(neg_p(X)).                8
                                                                    9
positive_q ← p(X).                                                  10
negative_q ← forall(X, neg_p(X)).                                   11
                                                                    12
positive_p(X) ← fail.                                               13
negative_p(X).                                                      14
```

Evaluation of query *neg_q* in program 5.5.4

```
?- neg_q.
   yes
?-
```

Example 5.5.3 The abductive problem is formed by Π, the abductive program 5.3.4, that declares $Abds = \{ r/1 \}$ and $IR = \{ \}$, and the query p.
The corresponding logic program is program 5.3.7, the result from converting the query p is $p([\,], AO)$, and its evaluation is:

Evaluation of the query $p([\,], AO)$ in program 5.3.7

```
?- p([], AO).
   AO = [r(1)] ? ;
   AO = [r(2)] ? ;
   no
```

In order to determine that the results obtained are correct, according to Def. 5.1.4, we have to test, for every solution, that

- for every term A, $A \in \Delta$, there are a term B, $B \in Abdcs$, and a (possible empty) substitution σ such that $A = B\sigma$ or $A = (\neg B)\sigma$.

  First solution: $r(1)$ is the only term in Δ and $r(X) \in Abds$ so, for $\sigma = X/1$, $A = B\sigma$.

  Second solution: $r(2)$ is the only term in Δ and $r(X) \in Abds$ so, for $\sigma = X/2$, $A = B\sigma$.

- $Q \in WFM\,(\Pi \cup IR \cup \Delta)$.

  First solution: the result of adding $r(1)$ to the original program is shown in program 5.5.5. It can be seen that $Q \in WFM\,(\Pi \cup IR \cup \Delta)$ by simply making the query p.

  Second solution: the result of adding $r(2)$ to the original program is shown in program 5.5.6. As the only differences with respect to program 5.5.5 are in the last two lines, we only show that ones. It can be seen that $Q \in WFM\,(\Pi \cup IR \cup \Delta)$ by simply making the query p.

- $\bot \notin WFM\,(\Pi \cup IR \cup \Delta)$. As $IR = \{ \}$, it does not need to be checked.

Program 5.5.5

```
p ← test_no_loops_on(p), positive_p.                                    1
neg_p ← test_no_loops_on(neg_p), negative_p.                            2
neg_p ← test_loop_on(neg_p),                                            3
        test_only_negative_literals_in_loop(neg_p).                     4
                                                                        5
positive_p ← q(X).                                                      6
negative_p ← forall(X, neg_q(X)).                                       7
                                                                        8
q(X) ← test_no_loops_on(q(X)), positive_q(X).                           9
neg_q(X) ← test_no_loops_on(neg_q(X)), negative_q(X).                   10
neg_q(X) ← test_loop_on(neg_q(X)),                                      11
        test_only_negative_literals_in_loop(neg_q(X)).                  12
                                                                        13
positive_q(1) ← r(1).                                                   14
positive_q(2) ← r(2).                                                   15
negative_q(X) ← X ≠ 1, X ≠ 2.                                           16
negative_q(1) ← neg_r(1).                                               17
negative_q(2) ← neg_r(2).                                               18
                                                                        19
r(X) ← test_no_loops_on(r(X)), positive_r(X).                           20
neg_r(X) ← test_no_loops_on(neg_r(X)), negative_r(X).                   21
neg_r(X) ← test_only_negative_literals_in_loop(neg_r(X)).               22
                                                                        23
positive_r(1).                                                          24
negative_r(X) ← X ≠ 1.                                                  25
```

Evaluation of query p in program 5.5.5

```
?- p.
   yes
?-
```

Program 5.5.6

```
...                                                                     22
                                                                        23
positive_r(2).                                                          24
negative_r(X) ← X ≠ 2.                                                  25
```

Evaluation of query p in program 5.5.6

```
?- p.
   yes
?-
```

Example 5.5.4 The abductive problem is formed by Π, the abductive program 5.3.4, that declares $Abds = \{ r/1 \}$ and $IR = \{ \}$, and the query *neg_p*.
The corresponding logic program is program 5.3.7, the result from converting the query *neg_p* is $neg\_p([\,], AO)$, and its evaluation is:

Evaluation of query $neg\_p([\,], AO)$ in program 5.3.7

```
?- neg_p([], AO).
   AO = [neg_r(1), neg_r(2)] ? ;
   no
```

In order to determine that the result obtained is correct, according to Def. 5.1.4, we have to test that

- for every term A, $A \in \Delta$, there are a term B, $B \in Abdcs$, and a (possible empty) substitution σ such that $A = B\sigma$ or $A = (\neg B)\sigma$. As Δ has two elements, we have to test it for both. $neg\_r(1) \in \Delta$ and $r(X) \in Abds$ so, for $\sigma = X/1$, $A = \neg B\sigma$. $neg\_r(2) \in \Delta$ and $r(X) \in Abds$ so, for $\sigma = X/2$, $A = \neg B\sigma$.
- $Q \in WFM\,(\Pi \cup IR \cup \Delta)$. The result from adding $neg\_r(1)$ and $neg\_r(2)$ to program 5.3.4 is program 5.5.7, where it can be seen that Q belongs to the WFM by simply making the query *neg_p*.
- $\bot \notin WFM\,(\Pi \cup IR \cup \Delta)$. As $IR = \{ \}$, it does not need to be checked.

Program 5.5.7

```
p ← test_no_loops_on(p), positive_p.                                  1
neg_p ← test_no_loops_on(neg_p), negative_p.                          2
neg_p ← test_loop_on(neg_p),                                          3
        test_only_negative_literals_in_loop(neg_p).                   4
                                                                      5
positive_p ← q(X).                                                    6
negative_p ← forall(X, neg_q(X)).                                     7
                                                                      8
q(X) ← test_no_loops_on(q(X)), positive_q(X).                         9
neg_q(X) ← test_no_loops_on(neg_q(X)), negative_q(X).                 10
neg_q(X) ← test_loop_on(neg_q(X)),                                    11
        test_only_negative_literals_in_loop(neg_q(X)).                12
                                                                      13
positive_q(1) ← r(1).                                                 14
positive_q(2) ← r(2).                                                 15
negative_q(X) ← X ≠ 1, X ≠ 2.                                         16
negative_q(1) ← neg_r(1).                                             17
negative_q(2) ← neg_r(2).                                             18
                                                                      19
r(X) ← test_no_loops_on(r(X)), positive_r(X).                         20
neg_r(X) ← test_no_loops_on(neg_r(X)), negative_r(X).                 21
```

```
neg_r(X) ← test_only_negative_literals_in_loop(neg_r(X)).        22
                                                                 23
positive_r(X) ← fail.                                            24
negative_r(1).                                                   25
negative_r(2).                                                   26
```

Evaluation of query *neg_p* in program 5.5.7

```
?- neg_p.
   yes
?-
```

Example 5.5.5 The abductive problem is formed by Π, the abductive program 5.2.1, that declares $Abds = \{ man/1, woman/1, parent/2 \}$ and $IR = \{ \bot \leftarrow man(X), woman(X) \}$, and the query $father(X, Y)$.
The corresponding logic program is:

Program 5.5.8

```
father(F, C, AI, AO) ←                                           1
      test_no_loops_on(father(F, C, AI, AO)),                    2
      positive_father(F, C, AI, AO).                             3
neg_father(F, C, AI, AO) ←                                       4
      test_no_loops_on(neg_father(F, C, AI, AO)),                5
      negative_father(F, C, AI, AO).                             6
neg_father(F, C, AI, AO) ←                                       7
      test_loop_on(neg_father(F, C, AI, AO)),                    8
      test_only_negative_literals_in_loop(                       9
                        neg_father(F, C, AI, AO)).               10
                                                                 11
positive_father(F, C, AI, AO)← man(F, AI, AA),                   12
                               parent(F, C, AA, AO).             13
negative_father(F, C, AI, AO)← neg_man(F, AI, AO) ;              14
                               neg_parent(F, C, AI, AO).         15
                                                                 16
mother(F, C, AI, AO) ←                                           17
      test_no_loops_on(mother(F, C, AI, AO)),                    18
      positive_mother(F, C, AI, AO).                             19
neg_mother(F, C, AI, AO) ←                                       20
      test_no_loops_on(neg_mother(F, C, AI, AO)),                21
      negative_mother(F, C, AI, AO).                             22
neg_mother(F, C, AI, AO) ←                                       23
      test_loop_on(neg_mother(F, C, AI, AO)),                    24
      test_only_negative_literals_in_loop(                       25
                        neg_mother(F, C, AI, AO)).               26
                                                                 27
positive_mother(M, C, AI, AO)← woman(M, AI, AA),                 28
                               parent(M, C, AA, AO).             29
```

```
negative_mother(M, C, AI, AO)← neg_woman(M, AI, AO) ;                30
                               neg_parent(M, C, AI, AO).             31
                                                                     32
man(X, AI, AO) ← add_abdcs([man(X)], AI, AO).                        33
neg_man(X, AI, AO) ← add_abdcs([neg_man(X)], AI, AO).                34
                                                                     35
parent(X, Y, AI, AO) ← add_abdcs([parent(X, Y)], AI, AO).            36
neg_parent(X, Y, AI, AO) ←                                           37
                 add_abdcs([neg_parent(X, Y)], AI, AO).              38
                                                                     39
                                                                     40
false(AI, AO) ← man(X, AI, AA), woman(X, AA, AO).                    41
neg_false(AI, AO) ←                                                  42
                  forall(X, ¬man(X); ¬woman(X), AI, AO).             43
```

The result from converting the query $Q = father(X,\ Y)$ is

$$father(X,\ Y,\ [\,],\ AO),\ neg\_false(AO,\ \_),$$

and its evaluation is:

Evaluation of query $father(X,\ Y,\ [\,],\ AO),\ neg\_false(AO,\ \_)$ **in program 5.5.8**

```
?- father(X, Y, [], AO), neg_false(AO, _).
   AO = [man(X), parent(X,Y)] ? ;
   no
```

In order to determine that the result obtained is correct, according to Def. 5.1.4, we have to test that

- for every term A, $A \in \Delta$, there are a term B, $B \in Abdcs$, and a (possible empty) substitution σ such that $A = B\sigma$ or $A = (\neg B)\sigma$. As Δ has two elements, we have to test it for both. $man(X) \in \Delta$ and $man(X) \in Abds$ so, for σ the empty substitution, $A = B\sigma$. $parent(X,Y) \in \Delta$ and $parent(X,Y) \in Abds$ so, for σ the empty substitution, $A = B\sigma$.

- $Q \in WFM\,(\Pi \cup IR \cup \Delta)$. The result from adding an instance of the predicates $man/1$ and $parent/2$ (the meaning of the abduction is that we need an instance of both and the second predicate has to use as first argument the only argument of the first predicate) to the program is program 5.5.9, where it can be seen that Q belongs to the WFM by simply making the query $father(X,Y)$

- $\bot \notin WFM\,(\Pi \cup IR \cup \Delta)$. $IR = \{\ \bot \leftarrow man(X),\ woman(X)\ \}$, and it can be checked by making the query "? $- false$.". As expected, it does not belong to the Well founded model of $(\Pi \cup IR \cup \Delta)$.

Program 5.5.9

```
abdcs([man/1, woman/1, parent/2]).                                    1
                                                                      2
father(F, C) ←                                                        3
      test_no_loops_on(father(F, C)),                                 4
      positive_father(F, C).                                          5
neg_father(F, C) ← test_no_loops_on(neg_father(F, C)),                6
                   negative_father(F, C).                             7
neg_father(F, C) ← test_loop_on(neg_father(F, C)),                    8
      test_only_negative_literals_in_loop(neg_father(F, C)).          9
                                                                      10
positive_father(F, C)← man(F), parent(F, C).                          11
negative_father(F, C)← neg_man(F) ; neg_parent(F, C).                 12
                                                                      13
mother(F, C) ← test_no_loops_on(mother(F, C)),                        14
               positive_mother(F, C).                                 15
neg_mother(F, C) ← test_no_loops_on(neg_mother(F, C)),                16
                   negative_mother(F, C).                             17
neg_mother(F, C) ← test_loop_on(neg_mother(F, C)),                    18
      test_only_negative_literals_in_loop(neg_mother(F, C)).          19
                                                                      20
positive_mother(M, C)← woman(M), parent(M, C).                        21
negative_mother(M, C)← neg_woman(M) ; neg_parent(M, C).               22
                                                                      23
false ← man(X), woman(X).                                             24
neg_false ← forall(X, ¬man(X); ¬woman(X)).                            25
                                                                      26
man(juan).                                                            27
parent(juan, pepe).                                                   28
```

Evaluation of the query $father(X,Y)$ to test that $Q \in WFM\,(\Pi \cup IR \cup \Delta)$:

Evaluation of query $father(X,Y)$ in program 5.5.9

```
?- father(X, Y), neg_false.
   F = juan
   C = pepe ? ;
   no
?-
```

Evaluation of the query $false$ to test that $\bot \notin WFM\,(\Pi \cup IR \cup \Delta)$:

Evaluation of query $false$ in program 5.5.9

```
?- false.
   no
?-
```

5.6 Summarizing the method of our adbuctive framework

In this chapter we have described our framework for abduction, capable of solving non-ground queries.

We started by defining our abductive problems as tuples $< \Pi, Abdcs, IR, Q >$ (Sec. 5.1), where Π is our abductive program, *Abdcs* is the set of abducibles, IR the set of integrity rules and Q the query. As our goal was to use logic programming to solve the queries, the translation of the abductive problems into logic programs was our next step (Sec. 5.2).

This translation is done in two steps:

- The syntax of the abductive programs Π was keep close to the syntax of logic programs to allow us to apply the transformation exposed in chapter 4. The result obtained from this step is an abductive program that is able to deal with negation with variables without suffering from floundering.
- In the previous step we still obtain an abductive program. Here this program is changed to obtain the abductions that explain the success of the query.

  This is done by:

  - adding the arguments needed to return and manage the abductions needed by the goals and subgoals.
  - creating for each abducible two new clauses that are in charge of abducing its positive and negative version, respectively.
  - translating the query so it fits to the new program, and guarantees that the IRs are satisfied.

During the translation process the following problems needed to be solved:

- The free variables in the bodies of the clauses need some kind of universal quantification management. As the one presented in Sec. 4.5 is not capable of returning the abductions that guarantee the success of the argument predicates for all the instantiations of its variables, in Sec. 5.3 we present a new implementation that is able to do it.
- The consistency of the set of abductions needs to be tested. This testing process needs to deal with the different forms of abductions that we have, and it is explained in detail in Sec. 5.4.

As a result from solving them two new forms of abductions appeared:

- the abduction with universally quantified variable(s) (see Def. 5.3.1) and
- the abduction with universally quantified and constrained variable(s) (see Def. 5.3.2)

CHAPTER 6

CONCLUSIONS

The work presented here solves the floundering problem of Well Founded Semantics (see Sec. 2.5) implementations as Global SLS-resolution [Prz89a, Ros92], SLG-resolution [CSW95, CW96] and SLT-resolution [SyYhY02] (a revision of them is done in chapter 3).

The achievement of this goal is done by combining a modified version of both the transformation presented in [MMNMH08] to obtain from a program its dual and the derivation procedure presented in [PAP$^+$91] to determine if a query belongs or not to the Well Founded Model (see Def. 2.5.2) of a program.

Both works were initially developed to be used in different environments:

- The dual program transformation was developed under a variant of Clark's Semantics called Kunen Semantics (see Sec. 2.2) and this semantics consider contradictory, inconsistent or incomplete information as a program error. This is not the case in the Well Founded Semantics, where this kind of information is seen as a knowledge undefinedness.

- The derivation procedure of [PAP$^+$91] was developed for ground programs, so variables are not considered and programs with variables need to be grounded before this procedure is applied, hindering the use of logic programming as a programming language.

So, when combining them some problems arise:

1. The dual program obtained from the transformation in [MMNMH08] uses inequalities (see Sec. 4.4) and universal quantification (see Secs. 3.5 and 4.5). While inequations are not affected by the supposed "erroneous programs", it is the universal quantification.

2. The derivation procedure from [PAP$^+$91] is only suitable for ground programs, and dual programs present the following characteristics:

    - they are not variable-free, so programs are not supposed to be ground.

- instead of having only rules for the positive literals (see Def. 2.1.8), now we have rules for both the positive and the negative literals.

The (summarized) solutions we propose are the following:

1. A new implementation of the universal quantification in which we determine if it holds by using abductive techniques. (see Secs. 3.5 and 4.5).

2. A new derivation procedure that treats positive and negative literals symmetrically and allows to make derivations with non-ground programs (see Sec. 3.4).

Besides, although modifications of the syntax transformation to get nonoverlapping clauses from overlapping ones and the implementation of inequalities are not strictly required for correctness in [MMNMH08], the following features motivated us for performing modifications:

- the definition of non-overlapping clauses (Def. 3.2.2) are based on testing two by two if they are compatible. We propose the definition of set of overlapping clauses (Def. 4.1.1), to improve the method. Although they are equivalent, instead of the existing loop in algorithm 4.1.1 that searches for an overlapping and transforms it, our algorithms 4.1.2 and 4.1.3 determine all the sets of overlapping clauses and algorithm 4.1.4 transforms them. From our definition it is possible to generate more compact and simpler programs.

- when an inequality has multiple different answers they try to simplify the solutions' set to avoid the existence of duplicates, and this is done by joining in a disjunctive way the multiple solutions. While this can be useful for some problems, the expected way of obtaining solutions when making a query in a Prolog interpreter is one by one. We propose a new inequalities implementation (see Sec. 4.4) that works in this way, enabling its use from other Prolog programs that are not capable of managing disjunctions of solutions.

With respect to the universal quantification, the method proposed in [MMNMH08] tests its satisfiability by checking that the affected formula holds for all the terms in the Herbrand Universe of the program. Apart from the fact that it is affected by the supposed "erroneous programs", checking one by one every term in the Herbrand Universe results very expensive computationally speaking. We propose a new method that relies on building tautologies from the solutions of the universally quantified variables (see Secs. 3.5 and 4.5). By making use of it, our implementation is able to solve problems like $\forall\ X.\ (\ X = 0 \vee X \neq 0\ )$, while the method in [MMNMH08] is not[1].

[1] Considering that the valid set of terms is the infinite set of the natural numbers, the formula can not be evaluated for all of them.

Comparing our negation system with the existing ones in Well Founded Semantics (SLS [Prz89a, Ros92], SLG [CSW95, CW96] and SLT [SyYhY02]), we found the following differences:

- these systems can not compute the answers to negative non-ground queries (in fact they restrict the use of variables) while our system is fully capable of doing it.

  Due to this, we can not compare which one has a better complexity when evaluating a non-ground query. Without regarding this, we must remark that, when the satisfiability of universal quantification does not need to be determined, the complexity is polynomial in the number of evaluated goals (as in SLS, SLT and SLG). The worst case occurs when it needs to be determined; in this case the complexity can be exponential in the number of solutions for the affected variables.

- with respect to ground negative queries, these systems need to compute that there is no proof for the positive query in order to determine that there exists one for the negative one.

  They do the computation of a query that might be more complex than the original one, while our negation system computes only the goals needed to determine if there is a proof for the goal.

- to solve the problem of infinite loops and redundant computations SLS and SLG make use of non-linear tabling, while SLT gains efficiency over them by using linear tabling (see Chapter 3).

  Our system uses a memorizing mechanism that is comparable to tabling only in the sense that it avoids loops (see Sec. 4.3).

  The drawbacks of this mechanism are that it is not capable of storing the necessary information to avoid redundant computations, delay the evaluation of infinite branches in the derivation tree until the finite ones have been evaluated, or compute fixed points to determine that no new solutions will be obtained if we continue evaluating an infinite branch. In Sec. 4.3 we expose the benefits of having this features, and illustrate them by means of some examples.

  It is clear that our implementation can be improved in this aspect (e.g. along the lines illustrated in section 4.3.1), but the modifications affect the Prolog engine, and our goal is to offer a new implementation of Well Founded Semantics that does not suffer from floundering when answering non-ground queries and works on any Prolog engine.

During the development of this work, we found ourselves using abductive techniques (see Chapter 5) to solve the problems of the inequalities implementation (see Sec. 4.4) and the universal quantification (see Secs. 3.5 and 4.5). In the former the inequalities are collected and stored as attributes of attributed variables (see Def. 4.4.1) and tested just after unification, while in the universal quantification the solutions of the affected variables are collected and the test is to build a tautology from them.

As we use abductive techniques to solve both the inequalities and the universally quantification problems, we found it interesting to use our implementation to develop an abductive framework. The existing abductive frameworks in logic programming can not deal with negative non-ground queries because they use negation systems that constraint the use of variables. As our system does not constraint the use of variables, we developed an extension of the negation system that is able to deal with non-ground abductive queries (see chapter 5).

The process performed by our abductive framework, in order to obtain a logic problem from the abductive problem, can be summarized into the following three steps:

- it makes use of the transformations performed by our negation system to obtain, from the abductive program, what should be called the abductive dual program,
- it makes some transformations to the abductive dual program it in order to obtain a logic program that, when answering a query, returns the hypotheses that justify the proof of the query (the abductive solutions),
- and it modifies the universal quantification implementation to return the hypotheses that explain its satisfaction.

The development of the new universal quantification lead us to the necessity of representing new forms of abductions: the ones with universally quantified variables (see Def. 5.3.1) and the ones with universally quantified variables and constrained (see Def. 5.3.2). Both of them are compact representations of the necessity of the abduction for all the possible values of the variables, but the second one eliminates the necessity of having the abduction for some values of the variable.

While their representation is only a matter of syntax, when checking the consistency of the abduction's set they need to be taken into account, and we present a method capable of determining its consistency even in presence of this compact representations of infinite sets (see Sec.5.4).

Summarizing, the results of this work are:

- an implementation of negation in Well Founded Semantics that does not suffer from the floundering problem when dealing with non-ground queries,
- and an abductive framework under Well Founded Semantics that deals with non-ground abductive queries.

Both can be downloaded from "*https://babel.ls.fi.upm.es/software/intneg-wfs/*".

BIBLIOGRAPHY

[AB94] Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: a survey. *JLP*, 19–20:9–72, July 1994.

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, 1988.

[APS04] José Júlio Alferes, Luís Moniz Pereira, and Terrance Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4), July 2004.

[Ass89] Association for Computing Machinery (ACM). *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania*. ACM Press, 1989. Chairman-Silberschatz, Avi.

[AvE82] Krzysztof R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, July 1982.

[BAK91] Roland N. Bol, Krzysztof R. Apt, and Jan Willem Klop. An analysis of loop checking mechanisms for logic programs. *Theor. Comput. Sci.*, 86(1):35–79, 1991.

[BD98] Roland N. Bol and Lars Degerstedt. Tabulated resolution for the well-founded semantics. *J. Log. Program.*, 34(2):67–109, 1998.

[BF91] Nicole Bidoit and Christine Froidevaux. General logical databases and programs: default logic semantics and stratification. *Inf. Comput.*, 91(1):15–54, 1991.

[BMPT87] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. Intensional negation of logic programs. *Lecture notes on Computer Science*, 250:96–110, 1987.

[BMPT90] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *JLP*, 8(3):201–228, 1990.

[Bry89] François Bry. Logic programming as constructivism: A formalization and its application to databases. In *PODS* [Ass89], pages 34–50. Chairman-Silberschatz, Avi.

[BS85] Genevieve Bossu and Pierre Siegel. Saturation, nonmonotonic reasoning and the closed-world assumption. *Artif. Intell.*, 25(1):13–63, 1985.

[Bue95] F. Bueno. *The CIAO Multiparadigm Compiler: A User's Manual*, 1995.

[CDT91] Luca Console, Daniele THESEIDER DUPRE, and Pietro Torasso. On the relationship between abduction and deduction, 1991.

[CH85] Ashok K. Chandra and David Harel. Horn clauses queries and generalizations. *J. Log. Program.*, 2(1):1–15, 1985.

[Cla78] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322, New York, NY, 1978. Plenum Press.

[CP86] Philip T. Cox and Tomasz Pietrzykowski. Causes for events: Their computation and applications. In Jörg H. Siekmann, editor, *CADE*, volume 230 of *Lecture Notes in Computer Science*, pages 608–621. Springer, 1986.

[CS92] L. Console and L. Saitta. Abduction, induction and inverse resolution. In *Proc. of the Compulog Workshop on Logic Programming and Artificial Intelligence, London, UK*, 1992.

[CS94] Eugene Charniak and Solomon Eyal Shimony. Cost-based abduction and map explanation. *Artif. Intell.*, 66(2):345–374, 1994.

[CSW95] Weidong Chen, Terrance Swift, and David Scott Warren. Efficient top-down computation of queries under the well-founded semantics. *J. Log. Program.*, 24(3):161–199, 1995.

[CW93] Weidong Chen and David Scott Warren. The slg system, 1993. Available by ftp from seas.smu.edu:pub or sbcs.sunysv.edu:pub/XSB.

[CW96] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *JOURNAL OF THE ACM*, 43:43–1, 1996.

[DK02] Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 402–436. Springer, 2002.

[DLV] DLV. The DLV home page. http://www.dbai.tuwien.ac.at/proj/dlv/.

[ECL] ECLiPSe. The ECLiPSe constraint logic programming language. http://www.eclipse-clp.org/, http://www-icparc.doc.ic.ac.uk/eclipse/, http://eclipse-clp.wiki.sourceforge.net/.

[Esh93] Kave Eshghi. A tractable class of abduction problems. In *IJCAI*, pages 3–8, 1993.

[Fit85] M. Fitting. A kripke-kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[Gel89] Allen Van Gelder. The alternating fixpoint of logic programs with negation. In *PODS* [Ass89], pages 1–10. Chairman-Silberschatz, Avi.

[Gel07] Michael Gelfond. *Handbook of Knowledge Representation (Foundations of Artificial Intelligence)*, chapter 1 (Answer Sets). Elsevier Science, December 2007.

[GG01] Hai-Feng Guo and Gopal Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In Philippe Codognet, editor, *ICLP*, volume 2237 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2001.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[GRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. The MIT Press, 1994.

[Hol92] Christian Holzbaur. Metastructures versus attributed variables in the context of extensible unification. In Maurice Bruynooghe and Martin Wirsing, editors, *PLILP*, volume 631 of *Lecture Notes in Computer Science*, pages 260–268. Springer, 1992.

[KK71] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.

[KKT92] Antonis C. Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *J. Log. Comput.*, 2(6):719–770, 1992.

[KM90] Antonis C. Kakas and Paolo Mancarella. Generalized stable models: A semantics for abduction. In *ECAI*, pages 385–391, 1990.

[Kow74] Robert Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[Kun87] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[Llo87] J. W. Lloyd. *Foundations of Logic Programing, 2nd edition*. Springer, 1987.

[LPF+06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.

[McC80] John McCarthy. Circumscription - a form of non-monotonic reasoning. *Artif. Intell.*, 13(1-2):27–39, 1980.

[Men96] Tim Menzies. Applications of abduction: Knowledge-level modeling. *International Journal of Human Computer Studies*, pages 325–0204, 1996.

[MH03] S. Muñoz-Hernández. *A Negation System for Prolog*. PhD thesis, Facultad de Informática (Universidad Politécnica de Madrid), 2003.

[MHMN00] S. Muñoz-Hernández and J.J. Moreno-Navarro. How to incorporate negation in a Prolog compiler. In V. Santos Costa E. Pontelli, editor, *2nd International Workshop PADL'2000*, volume 1753 of *LNCS*, pages 124–140, Boston, MA (USA), 2000. Springer.

[MHMNH01] S. Muñoz-Hernández, J.J. Moreno-Navarro, and M. Hermenegildo. Efficient negation using abstract interpretation. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning*, La Habana (Cuba), 2001.

[Min82] Jack Minker. On indefinite databases and the closed world assumption. In Donald W. Loveland, editor, *CADE*, volume 138 of *Lecture Notes in Computer Science*, pages 292–308. Springer, 1982.

[MMNMH08] Julio Mariño, Juan José Moreno-Navarro, and Susana Muñoz-Hernández. Implementing constructive intensional negation. *New Generation Comput.*, 27(1):25–56, 2008.

[Nai86] L. Naish. Negation and quantifiers in NU-Prolog. In *Proc. 3rd ICLP*, 1986.

[Naq86] Shamim A. Naqvi. A logic for negation in database systems. In Henry F. Korth, editor, *XP7.52 Workshop on Database Theory*, 1986.

[NSS00] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A system for answer set programming. In *8th. International Workshop on Nonmonotonic Reasoning*, April 2000.

[PAP+91] Luís Moniz Pereira, Joaquim N. Aparício, Lu'is Moniz Pereira, Joaquim N. Apar'icio, and José J. Alferes. Derivation procedures for extended stable models. In *In IJCAI'91*, pages 863–868. Morgan Kaufmann, 1991.

[Pei74] Charles S. Peirce. *Collected papers of Charles Sanders Peirce / edited by Charles Hartshorne and Paul Weiss*. Harvard University Press, Cambridge, Mass. :, 1974.

[Poo85] David Poole. On the comparison of theories: Preferring the most specific explanation. In *IJCAI*, pages 144–147, 1985.

[Poo89] David Poole. Explanation, prediction: an architecture for default, abductive reasoning. *Computational Intelligence*, 5:97–110, 1989.

[Poo93] David Poole. Probabilistic horn abduction and bayesian networks. *Artif. Intell.*, 64(1):81–129, 1993.

[Poo94] David Poole. A methodology for using a default and abductive reasoning system, 1994.

[Pop73] Harry E. Pople. On the mechanization of abductive logic. In *IJCAI*, pages 147–152, 1973.

[PP88] Halina Przymusinska and Teodor C. Przymusinski. Weakly perfect model semantics for logic programs. In *ICLP/SLP*, pages 1106–1120, 1988.

[PP90] Halina Przymusinska and Teodor Przymusinski. Semantic issues in deductive databases and logic programs. In *Formal Techniques in Artificial Intelligence*, pages 321–367. North-Holland, 1990.

[Prz88] Teodor C. Przymusinski. On the declarative semantics of (stratified) deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming.*, pages 193–216. Morgan Kaufmann, 1988.

[Prz89a] Teodor C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS* [Ass89], pages 11–21. Chairman-Silberschatz, Avi.

[Prz89b] Teodor C. Przymusinski. On the declarative and procedural semantics of logic programs. *J. Autom. Reasoning*, 5(2):167–205, 1989.

[Prz89c] Teodor C. Przymusinski. Three-valued formalizations of non-monotonic reasoning and logic programming. In *KR*, pages 341–348, 1989.

[RC94] R. Ramesh and Weidong Chen. A portable method of integrating slg resolution into prolog systems. In *SLP*, pages 618–632, 1994.

[Rei77] Raymond Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[Ros92] Kenneth A. Ross. A prodedural semantics for well-founded negation in logic programs. *J. Log. Program.*, 13(1):1–22, 1992.

[RRS+99] I. V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient access mechanisms for tabled logic programs. *The Journal of Logic Programming*, 38(1):31 – 54, 1999.

[Sat89] T. Sato. First order compiler: A deterministic logic program synthesis algorithm. *Journal of Symbolic Computation*, 8(6):605–627, 1989.

[SD94] Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *J. Log. Program.*, 19/20:199–260, 1994.

[She84] John C. Shepherdson. Negation as failure: A comparison of clark's completed data base and reiter's closed world assumption. *J. Log. Program.*, 1(1):51–79, 1984.

[She88] John C. Shepherdson. Negation in logic programming. In *Foundations of Deductive Databases and Logic Programming.*, pages 19–88. Morgan Kaufmann, 1988.

[SM91] T. Sato and F. Motoyoshi. A complete top-down interpreter for first order programs. In *Logic Programming, Proc. of the 1991 International Symposium*, pages 35–53. MIT Press, 1991.

[SSW96] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. An abstract machine for computing the well-founded semantics. In *JICSLP*, pages 274–288, 1996.

[ST84] T. Sato and H. Tamaki. Transformational logic program synthesis. In *Proc. of the International Conference on 5th Generation Computer Systems FGCS84*, pages 195–201, 1984.

[ST99] Fariba Sadri and Francesca Toni. Abduction with negation as failure for active databases and agents. In *Pitagora Editrice Bologna*, pages 353–362. Springer-Verlag, 1999.

[SW94] Terrance Swift and David S. Warren. Efficiently implementing slg resolution:, 1994.

[SyYhY02] Yi-Dong Shen, Li yan Yuan, and Jia huai You. Slt-resolution for the well-founded semantics. *Journal of Automated Reasoning*, 28:53–97, 2002.

[TK94] Francesca Toni and Robert A. Kowalski. Reduction of abductive logic programs to normal logic programs. In *Proc. 12th ICLP*, pages 367–381. MIT Press, 1994.

[TS86] Hisao Tamaki and Taisuke Sato. Old resolution with tabulation. In Ehud Y. Shapiro, editor, *ICLP*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986.

[VEK76a] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976.

[vEK76b] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[vG89] Alan van Gelser. Negation as failure using tight derivations for general logic programs. *J. Log. Program.*, 6(1-2):109–133, 1989.

[VGRS88] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 221–230, New York, NY, USA, 1988. ACM.

[Vie89] Laurent Vieille. Recursive query processing: The power of logic. *Theor. Comput. Sci.*, 69(1):1–53, 1989.

[Wan94] Pei Wang. From inheritance relation to non-axiomatic logic. *International Journal of Approximate Reasoning*, 11:281–319, 1994.

[War83] D.H.D. Warren. An abstract prolog instruction set, 1983. Technical Note 309. Artificial Intelligence Center, SRI International, Menlo Park CA.

[War92] David Scott Warren. Memoing for logic programs. *Commun. ACM*, 35(3):93–111, 1992.

[XSB] XSB. The XSB home page. http://www.cs.sunysb.edu/~sbprolog/xsb-page.html, http://xsb.sourceforge.net/.

[Zho94] Neng-Fa Zhou. Parameter passing and control stack management in prolog implementation revisited. *ACM Transactions on Programming Languages and Systems*, 18:752–779, 1994.