Semantics of structured normal logic programs

Edelmira Pasarella*,a, Fernando Orejas*, Elvira Pinoa, Marisa Navarrob

aDepartament de LSI, Universitat Politècnica de Catalunya, Jordi Girona, 1-3. 08034
Barcelona, Spain
bDepartamento de LSI, Universidad del País Vasco, Paseo Manuel de Lardizabal, 1, Apdo 649,20080, San Sebastián, Spain

Abstract

In this paper we provide semantics for normal logic programs enriched with structuring mechanisms and scoping rules. Specifically, we consider constructive negation and expressions of the form Q ⊃ G in goals, where Q is a program unit, G is a goal and ⊃ stands for the so-called embedded implication. Allowing the use of these expressions can be seen as adding block structuring to logic programs. In this context, we consider static and dynamic rules for visibility in blocks. In particular, we provide new semantic definitions for the class of normal logic programs with both visibility rules. For the dynamic case we follow a standard approach. We first propose an operational semantics. Then, we define a model-theoretic semantics in terms of ordered structures which are a kind of intuitionistic Beth structures. Finally, an (effective) fixpoint semantics is provided and we prove the equivalence of these three definitions. In order to deal with the static case, we first define an operational semantics and then we present an alternative semantics in terms of a transformation of the given structured programs into flat ones. We finish by showing that this transformation preserves the computed answers of the given static program.

Key words: semantics, normal logic programs, embedded implication, visibility rules, structuring mechanism, intuitionistic structures

1. Introduction

The semantics of logic programs is usually defined at three levels. The operational semantics is defined by means of a procedural mechanism that makes possible to infer correct conclusions from programs as answers to given queries.

*This work has been partially supported by the Spanish CICYT project FORMALISM (ref. TIN2007-66523) and by the AGAUR Research Grant ALBCOM (ref. SGR 20091137)
*Corresponding author

Email addresses: edelmira@lsi.upc.edu (Edelmira Pasarella), orejas@lsi.upc.edu (Fernando Orejas), pino@lsi.upc.edu (Elvira Pino), marisa.navarro@ehu.es (Marisa Navarro)
The logical semantics studies programs as logical theories, defining the meaning of a program in terms of a class of models in the underlying logic. Finally the algebraic semantics defines the meaning of a logic program $P$ in terms of a certain intended model. This intended model is usually (effectively) computed by a bottom-up construction which is the least fixpoint of a function, called immediate consequence operator, defined over the set of the considered interpretations organized as a complete partially ordered set. The study of each of these semantics has its own interest. The operational semantics is the basis for the implementation of the language. The logical semantics allows us to reason about a given program. Finally, the algebraic semantics is usually the basis for the construction of a number of analysis tools. A very desirable property for any class of logic programs is what we may call the equivalence of the three semantic definitions. More precisely, this equivalence means, on the one hand, the soundness and completeness of the operational semantics with respect to the algebraic and the logical semantics, and, on the other hand, that the algebraic semantics is a typical model of the class defined by the logical semantics. For instance, for the class of (definite) Horn logic programs, the operational semantics corresponds to SLD-resolution [1], the logical semantics is defined in terms of the first-order theory associated to the program, and the intended model of a program $P$ is its least Herbrand model. Moreover, this model is effectively computed as the least fixpoint of an immediate consequence operator defined on Herbrand structures (see, e.g. [20, 38]). Another model-theoretic characterization of logic programs is given by the s-semantics approach [5] in which the intended model is closer to the operational semantics.

In order to structure logic programs there are two main approaches (see [6] for a survey). The first one is based on defining some notion of program unit or module and on providing a number of composition operators. Basically, this approach is oriented towards programming-in-the-large. The second approach consists in enriching logic programs with an abstraction mechanism and scoping rules, similar to block structuring, as found, for instance, in procedural programming. Hence, this approach seems to be suitable for programming-in-the-small in a structured manner. More precisely, this approach has been advocated by Miller [26] and others using (intuitionistic) implications embedded in the goals of programs as the structuring mechanism. In particular, Miller considered a class of programs close to Harrop formulas [27] in which, in addition to embedded implication in goals, disjunctive goals and explicitly quantified goals are admitted. As far as we know, the use of implications in goals was first advocated by Gabbay and Reyle [13] who proposed an extension of Prolog, called N-Prolog. Their aim was to extend Prolog with a mechanism that could allow them to add additional clauses to programs. That is, a mechanism that could implement hypothetical reasoning. Due to this initial approach, embedded implication is also known as hypothetical implication.

Extending logic programs with embedded implication means having expressions of the form $Q \supset G$ in goals, where $Q$ is a set of program clauses, $G$ is a goal and $\supset$ stands for the embedded implication. The intuitionistic semantics of this connective means that $G$ will hold if it can be proved with the help of the
clauses in \( Q \). This can be seen as adding block structuring to logic programs or, equivalently, as adding a \textit{where} construct to logic programs, since the clauses in \( Q \) may be seen as auxiliary (local) definitions to be used in the evaluation of \( G \).

In addition, the use of embedded implications allows a programmer to reuse some programs, when some extra conditions have to be considered. For example, let us consider a banking application, where a predicate \textit{loan}(C,N) states if it is possible to give a loan of N dollars to a customer \( C \). The definition of this predicate may be relatively complicated. It may have to consider what kind of customer is \( C \) (for example \( C \) it is a standard customer or a preferred customer), the value of his properties, and his previous history. Now, suppose that we want to define a new predicate, \textit{special loan}(C,N), for a special kind of loan, whose difference with \textit{loan}(C,N) is that shareholders of the bank are considered similarly to preferred customers. The definition of this predicate is very simple using an embedded implication:

\[
\textit{special loan}(C,N) \leftarrow \{\text{pref customer}(C) \leftarrow \text{shareholder}(C)\} \supset \text{loan}(C,N)
\]

Giordano, Martelli and Rossi [17] noticed that the semantics of logic programs would change depending if we interpret embedded implications following a static or a dynamic visibility rule, as done in procedural languages. Let us show a simple example borrowed from [6]. Suppose that we have the program \( P = \{p \leftarrow q\} \) and the query \( Q \supset p \), where \( Q = \{q\} \). According to [26], answering this query means seeing if we can satisfy \( p \) using the clauses in \( P \) together with the local definitions of the goal, i.e. \( \{q\} \). This means checking \( P \cup \{q\} \vdash p \). Obviously, the answer is that the goal is satisfiable. However, in [17] it is argued that this interpretation may be considered inadequate, if we want that our structuring mechanism resembles block structuring in most procedural languages. In particular, in these languages block structuring is based on what is called \textit{static visibility}. This means that local definitions are only visible at their definition scope, but not at an external or a more global scope. For instance, if a block \( Q \) includes a certain local definition and another block \( P \) either includes \( Q \) or is external to \( Q \), then that definition would not be visible in \( P \). Now, thinking of the resolution process associated to our example, to solve the goal \( p \) we cannot apply any clause from \( Q \). We can only apply a resolution step with clause \( p \leftarrow q \) from \( P \), obtaining the goal \( q \). Now we have to solve \( q \). However, the local definitions in \( Q \) are considered invisible in \( P \). Hence, if the principle of static visibility has to be applied, we would be unable to use the clause \( q \) to complete the resolution process. Giordano, Martelli and Rossi also argued that the use of Miller’s approach could be considered equivalent to having dynamic visibility. This means that a local definition can be used if the current execution allows it. For instance, in our example, we would be allowed to resolve the goal \( q \) using the local definition in \( Q \) when we are in the process of resolving the goal \( Q \supset p \), but otherwise it would not be allowed.

As a consequence, in [17], the authors considered that the choice of one of these two interpretations of the embedded implication is just an option of the language designer, though it should be clear that, depending of the choice, the
semantics of a program will differ. In particular, it is considered that dynamic visibility is better suited for hypothetical reasoning, while the static rule is more convenient for structuring logic programs into blocks. Following these ideas, in the present paper, we study both cases.

Most previous work on this topic only applies to the use of embedded implication in definite logic programs, where negation is not considered. As far as we know, all the existing approaches studying the extension of normal logic programming with embedded implication consider negation as finite failure (perhaps with some other restrictions) and dynamic visibility. From our point of view, the major shortcoming of negation as failure concerns the limitations of this mechanism to deal with the whole class of normal programs (due to floundering). Its more direct extension is constructive negation, introduced by Chan [7, 8] for the case of Datalog programs, and by Stuckey [35] and Drabent [9] for the case of general programs, which is sound and complete with respect to the Clark-Kunen completion [19] for the whole class of normal programs. As a consequence, our semantics is based on constructive negation.

Other approaches to the semantics of negation in logic programming concentrate on defining the most appropriate algebraic semantics for a given program, not dealing especially with operational issues (actually, some of these semantics are not recursively enumerable for the general case). In particular, the best known of these approaches define the intended models of a given program as stable models [15], perfect models [33], or well-founded models [14]. Our choice of semantics, in terms of the operational rule of constructive negation, is based on the nature of our research. More precisely, the standard semantic approach to embedded implications is based on extending the given operational semantics (SLD resolution in the standard case of definite programs) by rules that describe how the local definitions are used in a given computation. Then, a corresponding (intuitionistic) model theory is defined according to this operational semantics. In this sense, we have followed a similar path. We have extended the most general operational semantics for negation with the rules to handle embedded implications and, then, we have defined a corresponding intuitionistic model theory. What we consider interesting, in our case, is that the intuitionistic models not only take care of the embedded implications, but also of the negation. Nevertheless, it may be interesting to study what would be the corresponding notion of (intuitionistic) well-founded models for normal logic programs with embedded implications.

The two approaches that we use to define the semantics of programs for the two kinds of visibility rules are very different. In particular, we approach the semantics of programs with dynamic visibility in the standard way: we define operational, logical, and algebraic semantics and show their equivalence. More precisely, the operational semantics is a combination of the semantics of constructive negation [9] and the semantics of the embedded implications [26]. The logical semantics is defined in terms of a class of Beth models and the algebraic semantics is defined in terms of an immediate consequence operator defined on classes of these (intuitionistic) Beth models.

In the case of logic programs with static visibility, the operational seman-
tics is also a combination of the semantics of constructive negation [9] and the semantics of the embedded implications [17], but the logical and the algebraic semantics are defined indirectly, by means of a translation into programs not including embedded implications. Nevertheless, we prove the soundness and completeness of the operational semantics of a program with respect to the logical and algebraic semantics of its translation. One may wonder why not using the same kind of approach for dealing with the two kinds of visibility rules. The main reason is that the other possibilities seemed more complex than needed. In particular, in the case of the dynamic visibility rule, we did not see a reasonable way of defining the semantics of a program via a translation, unless we imposed some kind of restriction on the programs, like stratification, as done in [11]. Conversely, all our attempts to define directly a model-theoretic semantics for programs with the static rule resulted in a very complex definition. Anyhow, we believe that each approach has its own advantages and inconveniences. On the one hand, we believe that a direct semantics provides better insights than a translation semantics about the constructions being studied. Conversely, a translation semantics can directly be the basis for an implementation.

We consider that the paper has two main contributions. The first one is the model-theoretic semantics for normal programs with embedded implications and dynamic visibility, which can be seen as a solution to a problem that, in a way, is open since 1984, when Gabbay presented the first semantic approach [12] to define a model-theoretic semantics for this class of programs. The second contribution is the transformational semantics for this class of programs, when considering static visibility. We think that this contribution is interesting since it can be the basis for a simple implementation of these constructions.

The results presented in this paper can be seen as a more coherent, simplified and detailed presentation of the results presented in [29] and [30].

The paper is organized as follows. Section 2 contains some basic notions and notational conventions, including a small introduction to intuitionistic models (especially Beth models). In sections 3 and 4 we provide semantics for the class of dynamic normal logic programs and static normal logic programs, respectively. Then, in Section 5 we review related work. In Section 6 we provide some conclusions and discuss further work. Finally, to enhance readability, an appendix includes the proofs of the main results.

2. Basic definitions and notation

In this section we introduce some basic concepts and notation that are used along the paper. More precisely, in the first subsection we introduce some basic logical notation and terminology. Then, in the second subsection we describe the syntax of the programs that we consider in the paper. Finally, the last subsection is a brief introduction to intuitionistic structures, used to give a model-theoretic semantics to our programs, when considering dynamic visibility.
2.1. Basic notation

A signature $\Sigma$ consists of a pair of sets $(FS_\Sigma, PS_\Sigma)$ of function and predicate symbols, respectively, with some associated arity. $T_\Sigma(X)$ denotes the set of all $\Sigma$-terms over variables from $X$. Given $p \in PS_\Sigma$ and $\Sigma$-terms $t_1, \ldots, t_n$, a literal is either an atom $p(t_1, \ldots, t_n)$ (namely a positive literal) or a negated atom $\neg p(t_1, \ldots, t_n)$ (namely a negative literal). The set $Form_\Sigma$ consists of all $\Sigma$-formulas, which are written (from atoms) using connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, and quantifiers $\forall, \exists$. Given $\varphi \in Form_\Sigma$, we denote by $\text{free}(\varphi)$ the set of all free variables occurring in $\varphi$. $\varphi(\overline{\tau})$ specifies that $\text{free}(\varphi) \subseteq \overline{\tau}$. $S_{\text{eq}}\Sigma$ is the set of all $\Sigma$-sentences, i.e., formulas $\varphi \in Form_\Sigma$ such that $\text{free}(\varphi) = \emptyset$. By $\varphi^\forall_{\overline{\tau}}$ (resp. $\varphi^\exists_{\overline{\tau}}$) we denote the formula $\forall x_1 \ldots \forall x_n(\varphi)$ (resp. $\exists x_1 \ldots \exists x_n(\varphi)$), where $x_1, \ldots, x_n$ are the variables in $\text{free}(\varphi) \setminus \overline{\tau}$. In particular, the universal (resp. existential) closure, that is $\varphi^\forall^{\emptyset}$ (resp. $\varphi^\exists^{\emptyset}$) is denoted by $\varphi^\forall$ (resp. $\varphi^\exists$). $\text{true}$ represents the logical constant. The set of naturals is denoted by $\mathbb{N}$. In general, subscripts and superscripts will be used if needed and an overline over an object is used to denote (finite) sequences of that kind of object.

**Definition 2.1.** Given a signature $\Sigma$, the free-equality theory, $FET_\Sigma$, can be presented by the following axioms:

1. $\forall x(x = x)$
2. $\forall \forall \forall \forall (\vec{x} = \vec{y} \iff f(\vec{x}) = f(\vec{y}))$ for each $f \in FS_\Sigma$
3. $\forall \forall \forall \forall (\vec{x} = \vec{y} \rightarrow (p(\vec{x}) \iff p(\vec{y})))$ for each $p \in PS_\Sigma \cup \{=\}$
4. $\forall \forall \forall \forall (f(\vec{x}) \neq g(\vec{y}))$ for each $f, g \in FS_\Sigma$, with $f$ different from $g$
5. $\forall x(x \neq t)$ for each $\Sigma$-term $t$ and variable $x$ such that $x \in \text{var}(t)$ and $x$ is different from $t$

Whenever $\Sigma$ has a finite number of function symbols it is necessary to add the (Weak) Domain Closure Axiom (WDCA):

$$\forall x \left( \bigvee_{f \in FS_\Sigma} \exists y_1, \ldots, y_{n_f} (x = f(y_1, \ldots, y_{n_f})) \right)$$

to $FET_\Sigma$ in order to make it a complete theory, that is, $FET_\Sigma \models \varphi$ or $FET_\Sigma \models \neg \varphi$ for any $\Sigma$-sentence.

**Definition 2.2.** An equality $\Sigma$-constraint, constraint for short when $\Sigma$ is implicit, is a $\Sigma$-formula where the only predicate symbol occurring in atoms is the equality. A constraint $c$ is satisfiable if $FET_\Sigma \models c^\forall$. Given constraints $c$ and $d$, $c$ is more general than $d$ if $FET_\Sigma \models (d \rightarrow c)^\forall$, and $c$ and $d$ are equivalent if $FET_\Sigma \models (d \leftrightarrow c)^\forall$.

2.2. Syntax

We consider normal constraint logic programs extended with embedded implications. This extension actually affects the definition of literals and goals in which embedded implication may occur. We refer to these syntactic objects as
extended literals and extended goals, respectively. In the following we assume the existence of an underlying signature $\Sigma = (FS_\Sigma, PS_\Sigma)$.

**Definition 2.3.** Programs, clauses, (extended) literals and goals are defined using the BNF presentation in Figure 1, where $A$ and $C$ are syntactic variables which range over $\Sigma$-atoms and $\Sigma$-constraints, respectively. Moreover, $G$, $L$, $K$ and $P$ range over extended $\Sigma$-goals, lists of $\Sigma$-goals, $\Sigma$-clauses and $\Sigma$-programs, respectively.

\[
G ::= \text{true} | A | \neg A | P \supset G \\
L ::= G | G, L \\
K ::= A \leftarrow L \bowtie C \\
P ::= K | K; P
\]

Figure 1: Clauses and extended goals

From the previous definition we have the following:

- A $\Sigma$-program is considered to be a non-empty finite set of $\Sigma$-clauses, even if in the BNF definition programs could be seen as sequences of clauses separated by a semicolon.

- An extended $\Sigma$-goal is either the logical constant $\text{true}$; or a normal $\Sigma$-literal, $p(\vec{x})$ or $\neg p(\vec{x})$; or an expression of the form $P \supset G$, where $P$ is a $\Sigma$-program and $G$ is an extended $\Sigma$-goal.

- A list of extended $\Sigma$-goals is a non-empty sequence of extended $\Sigma$-goals.

- A $\Sigma$-clause is an expression of the form $p(x_1, \ldots, x_n) \leftarrow G_1, \ldots, G_k \bowtie c$ with $k \geq 1$. where $p \in PS_\Sigma$ with arity $n \geq 0$; $x_1, \ldots, x_n$ are variables; $G_1, \ldots, G_k$ is a list of extended $\Sigma$-goals; and $c$ is a $\Sigma$-constraint. As usual in constraint logic programming, we assume that the heads in $\Sigma$-clauses are flat (i.e. their only subterms are variables) and we separate literals from constraints in the body of clauses (using the symbol $\bowtie$). It may be noted that using flat heads is not a real limitation, since a clause of the form $p(t_1, \ldots, t_n) \leftarrow G_1, \ldots, G_k \bowtie c$, with $k \geq 1$, $n \geq 0$, where $t_1, \ldots, t_n$ are $\Sigma$-terms, is equivalent to the constrained clause $p(x_1, \ldots, x_n) \leftarrow G_1, \ldots, G_k \bowtie c \land x_1 = t_1 \land \ldots \land x_n = t_n$.

- Additionally, throughout the rest of the paper, we adopt the following conventions: $P$ and $Q$ denote $\Sigma$-programs. Clauses may be embraced in parenthesis. Instead of using the syntactic variable $L$, we denote lists of extended $\Sigma$-goals by $\overline{G}$. In a $\Sigma$-clause $p(x_1, \ldots, x_n) \leftarrow \overline{G} \bowtie c$, if the $\Sigma$-constraint $c$ is $\text{true}$ we simply write $p(x_1, \ldots, x_n) \leftarrow \overline{G}$ and if the $\Sigma$-goal $\overline{G}$ and the $\Sigma$-constraint $c$ are both $\text{true}$ we simply write $p(x_1, \ldots, x_n)$. 


Example 2.1. To illustrate the syntax let us consider the following simple program:

\[ P = r(x) \leftarrow \{ (p(y) \leftarrow \neg q(y)) \} \supset s(x) \]
\[ s(x) \leftarrow p(x) \]

Scope of variables. As usual in logic programming, we assume that free variables in a clause are implicitly universally quantified. More precisely, given a clause \( p(\overline{x}) \leftarrow G_1, \ldots, G_k \square c, \) we consider that the set of free variables of the clause is the union of the sets of free variables of \( p(\overline{x}), G_1, \ldots, G_k, \) and \( c, \) where the set of free variables of a goal \( P \supset G \) consists just of the free variables of the goal \( G. \) This means that the scope of a variable is the clause where it is (implicitly) defined, and it is not visible outside that clause.

For example, the clause

\[ r(x) \leftarrow \{ (p(y) \leftarrow \neg q(y)) \} \supset s(x) \]

is interpreted as

\[ \forall x \left( r(x) \leftarrow \{ \forall y (p(y) \leftarrow \neg q(y)) \} \supset s(x) \right) \]

and the clause

\[ r(x) \leftarrow \{ (p(x) \leftarrow \neg q(x)) \} \supset s(x) \]

is interpreted as

\[ \forall x \left( r(x) \leftarrow \{ \forall x (p(x) \leftarrow \neg q(x)) \} \supset s(x) \right) \]

which is equivalent to the clause

\[ \forall x \left( r(x) \leftarrow \{ \forall y (p(y) \leftarrow \neg q(y)) \} \supset s(x) \right) \]

Definition 2.4. The set of definitions of a predicate \( p \) with respect to a program \( P \) is given by:

\[ \text{Def}(P,p) = \{(p(\overline{x}) \leftarrow \overline{G} \square c) \in P\} \]

For the sake of simplicity, in the following, we omit the prefix \( \Sigma- \) when it is clear from the context.

2.3. Intuitionistic structures

There are two well-known kinds of intuitionistic structures, Kripke structures and Beth structures. Kripke structures are quite well-known in computer science, because they are also used as models for temporal logics and other kinds of modal logics, but Beth structures are much less known. However, both kinds of structures are quite similar. As a consequence, we think that, following van Dalen [37], introducing Beth structures as a variation of Kripke structures may help many readers to get a good understanding.
The basic difference between intuitionistic and classical logic is that in intuitionistic logic all proofs must be constructive. For instance, it is not enough to prove the existence of an element satisfying a certain property by showing the impossibility of its inexistence. This difference applies also to the meaning of negation or implication. In particular, in classical logic, to prove that $A$ implies $B$ we have to show that either $\neg A$ or $B$ hold. However, in intuitionistic logic to prove that $A$ implies $B$ we must show that, in some sense, the truth of $B$ depends on the truth of $A$. In proof-theoretic approaches, this is formalized by saying that a proof of $B$ can be built using a proof of $A$ or, in a similar sense, considering that a proof of an implication is a function that given a proof of $A$ returns a proof of $B$. In model-theoretic approaches (like Beth’s and Kripke’s) this intuition can be explained in terms of two basic ideas. The first one is the notion of world that may be considered to represent the “knowledge” that we have at a certain moment. Technically, a world is seen as a first-order structure. The second idea is that a formula can be considered to hold if we can infer its truth from the knowledge that we have (or that we may acquire in the “future”). For instance, we can consider that the implication $A \supset B$ holds if, whenever our knowledge tells us that $A$ holds, it also tells us that $B$ holds.

Following these ideas, an intuitionistic structure $\mathcal{S}$ is a triple $\langle W, \preceq, I \rangle$ where

i. $(W, \preceq)$ is a partially ordered set of worlds. Each world represents, as said above, the knowledge that we may have at a given moment, and the ordering is related to the amount of knowledge associated to each world. This means that if $v$ and $w$ are two worlds in $W$ and $v \preceq w$ then $w$ includes more knowledge than $v$.

ii. $I$ is an interpretation function that maps every world $w \in W$ into a first-order structure $I(w)$. In particular, we see first-order structures $I(w)$ as sets of atomic formulas that represent the knowledge that we have in that world. Moreover, $I$ is monotonic in the sense that, $v \preceq w$ implies $I(v) \subseteq I(w)$.

Satisfaction of formulas in an intuitionistic structure is defined in terms of a forcing relation, denoted $\models$, that describes when a formula can be assumed to hold in a given world. A simple way of defining forcing could be to consider, at least for atomic formulas, that it coincides with satisfaction in the interpretation of the given world, i.e.:

$$v, \mathcal{S} \models \phi \quad \text{if} \quad I(v) \models \phi$$

Actually, this is the definition of forcing for atomic formulas in Kripke structures. However, in the case of Beth structures, the definition of forcing is a bit more involved. The basic idea is to consider that a formula is forced in a given world not only if the formula holds for the given knowledge included in this world (i.e. its interpretation) but also if we know that it must hold in the future. More precisely, first, we may consider that a (possibly infinite) ascending sequence of worlds $w_0 \preceq w_1 \preceq \ldots \preceq w_i \ldots$ represents a possible way of completing the knowledge that we have in the world $w_0$. In the same sense we
may consider that all the maximal ascending sequences (called paths) starting at \( w_0 \) represent all the possible ways to complete the knowledge that we have in the world \( w_0 \). Then, we may say that a formula is forced to hold in a given world \( w_0 \) if it is satisfied in all the possible ways of completing the knowledge in \( w_0 \). This can be formalized as follows:

Given a world \( w_0 \), we say that a set of worlds \( B \subset W \) is a bar for \( w_0 \) if for every path beginning with \( w_0 \) there is a world \( w_i \) in that path such that \( w_i \in B \).

Then, the forcing relation for Beth structures is defined as follows:

1. If \( \phi \) is atomic, \( v, S \models \phi \) if there is a bar \( B \) for \( v \) such that for all \( w \in B \), \( I(w) \models \phi \).
2. \( v, S \models \phi \land \psi \) if \( v, S \models \phi \) and \( v, S \models \psi \).
3. \( v, S \models \phi \lor \psi \), if there is a bar \( B \) for \( v \) such that for all \( w \in B : w, S \models \phi \) or \( w, S \models \psi \).
4. \( v, S \models \phi \supset \psi \) if for all \( w, \ w \geq v \ : \ w, S \models \phi \) then \( w, S \models \psi \).
5. \( v, S \models \forall x \phi \) if for every substitution \( \sigma \), \( w, S \models \sigma(\phi) \).
6. \( v, S \models \exists x \phi(x) \) if there is a bar \( B \) for \( v \) such that for all \( w \in B \) there is a substitution \( \sigma \) such that \( w, S \models \sigma(\phi) \).
7. \( v, S \models \neg \phi \), if for all \( w, \ w \geq v \ : \ w, S \not\models \phi \).

Finally, for the given notion of forcing, we consider that a Beth structure is a model of a closed formula \( \phi \) if \( v, S \models \phi \) for all \( v \in W \).

3. Semantics of dynamic normal logic programs

As explained in the introduction, the intuition of the dynamic interpretation of embedded implications is that, given a program \( P \), to prove the query \( Q \supset G \) it is necessary to prove \( G \) with the program \( P \cup Q \). This is formalized in [26] using the following inference rule:

\[
P \cup Q \models_{dyn} G \\
P \models_{dyn} Q \supset G
\]

Figure 2: Dynamic rule (D-rule)

In [26], both implications, clausal implication and embedded implication, are interpreted as intuitionistic implications. In particular, a main result of Miller's approach is that the proof-theoretic semantics for this kind of programs can be given in terms of intuitionistic logic. Moreover, he proposed a model-theoretic semantics in terms of Kripke models.

The aim of this section is to extend this approach to the case of programs that also include negation. In the first subsection we introduce an operational semantics for the class of normal logic programs with embedded implication. This semantics is an extension of constructive negation with the above rule to handle implication goals. It may be noted that our semantics is relatively
simple, but not immediately useful for practical purposes, since it is too non-deterministic to be directly implemented. Its main aim is to show the adequacy of the model-theoretic semantics defined below. In particular, our treatment of negation for standard goals, as shown in rules 1) and 2) of Definition 3.1, is a slightly simpler, but equivalent, variation of constructive negation as defined in [9]. Instead, we could have introduced an operational semantics closer to implementation, but the proofs for the main results of this section would have been slightly more complex.

Extending similarly the model-theoretic semantics, both to define the logical semantics and the least fixpoint semantics of a program, is quite more involved. In particular, a main difficulty comes from the non-monotonic nature of negation in logic programming which does not fit well with modularity. In [22], a new declarative compositional semantics was defined for a general class of normal logic program units, in terms of a class of models called ranked. As it was pointed out in that paper, ranked models are, intuitively, quite close to Beth models. This lead us to think that both connectives could have a natural and reasonably simple semantics in terms of intuitionistic (Beth) models. Moreover, this semantics would make more explicit the intuitionistic nature of negation in logic programming already pointed out by other authors (e.g. [34]).

Following these ideas, in Section 3.2 we define the Beth models that capture our intuition, together with their associated forcing relation. Then, in Section 3.3 we introduce an immediate consequence operator showing that it is monotonic and continuous. Moreover, in Section 3.4, we show that the least fixpoint of this operator coincides with the least model of the given program. Finally, in Section 3.5, the operational semantics is proved to be sound and complete with respect to the least fixpoint semantics.

### 3.1. Operational semantics

In this section we introduce an operational semantics for the class of normal logic programs with embedded implication. This semantics is presented in terms of a derivation relation over sequents of the form $P \vdash_{dyn} G \square c$, where $P$ is a $\Sigma$-program and $G \square c$ is a $\Sigma$-goal.

**Definition 3.1.** The derivation relation $\vdash$ over sequents is defined as follows:

1. $P \vdash_{dyn} G_1, p(\pi), G_2 \square c \vdash P \vdash_{dyn} G_1, G, G_2 \square c \land d$ if there exists a (renamed apart) clause $p(\pi) \leftarrow G_1, \ldots, G_m \square d \in Def(P, p)$ and $FET_{\Sigma} \models (c \land d)^3$

2. $P \vdash_{dyn} G_1, \neg p(\pi), G_2 \square c \vdash P \vdash_{dyn} G_1, G_2 \square c'$ if for every (renamed apart) clause $p(\pi) \leftarrow G_1, \ldots, G_m \square d \in Def(P, p)$, there exists $J \subseteq \{1, \ldots, m\}$, not necessarily unique, such that for each $j \in J$, assuming $G_j = Q_1 \supset \ldots \supset Q_k \supset \ell$ with $k \geq 0$, we have $P \cup Q_1 \cup \ldots \cup Q_k \vdash_{dyn} \neg \ell \square d \leftarrow P \cup Q_1 \cup \ldots \cup Q_k \vdash_{dyn} \square d_j$, and $FET_{\Sigma} \models (c' \leftarrow \neg d \lor \bigvee_{j \in J} d_j)^\gamma$, where $\leftarrow$ is the transitive closure of $\vdash$.

3. $P \vdash_{dyn} G_1, Q \supset G, G_2 \square c \vdash P \vdash_{dyn} G_1, G_2 \square c'$ if $P \cup Q \vdash_{dyn} G \square c \leftarrow P \cup Q \vdash_{dyn} G \square c'$. 

11
Definition 3.2. Let $P$ be a $\Sigma$-program and $G \Box c$ a $\Sigma$-goal. We say that $P \vdash_{dyn} G \Box c$ can be dynamically proved with computed answer $c'$, denoted $P \vdash_{dyn} G \Box c \Rightarrow_{dyn} c'$, if $P \vdash_{dyn} G \Box c \leadsto P \vdash_{dyn} \Box c'$, $FET_\Sigma \models c^3$ and $FET_\Sigma \models (c' \rightarrow c)^3$.

Intuitively Definition 3.1.2 means that a negative goal $\neg p(x)$ can be proved with computed answer $c'$ whenever it is possible to obtain contextual failures in the body of every definition of predicate $p(x)$. This means that it is enough to calculate the computed answers of the literals stated by $J$ in such a way that the solutions of the constraint $c'$ are included in the solutions of $\neg d$ and the union of the solutions of the corresponding computed answers. In what follows, for simplicity, whenever a constrained $\Sigma$-atom $\neg p(x) \Box c$ occurs in the right-hand side of a sequent, we denote it by $p(x) \Box c$. Next example illustrates these notions.

Example 3.1. Given the following programs:

$$P = \{ (p(x) \leftarrow p(x) \Box x = a); (q(x) \leftarrow \Box x = a) \}$$

$$Q = \{(r \leftarrow p(x), \neg q(x))\}$$

We want to prove that $P \vdash_{dyn} Q \supset r \leadsto P \vdash_{dyn} \Box true$, which means that, according to Definition 3.1.3, we have to prove that $P \cup Q \vdash_{dyn} \neg r \leadsto P \cup Q \vdash_{dyn} \Box true$. To see this, we use Definition 3.1.2, where the only clause defining $r$ is $(r \leftarrow p(x), \neg q(x))$. Let $G_1 = p(x)$ and $G_2 = \neg q(x)$, we have to see that there exists a subset $J \subseteq \{1, 2\}$ such that for every $j \in J$, $P \vdash_{dyn} \neg G_j \Box true \Rightarrow_{dyn} d_j$, since in this case $d = true$. Let $J = \{1, 2\}$, we have:

- For $G_1$, we have to prove $P \cup Q \vdash_{dyn} \neg p(x) \leadsto P \cup Q \vdash_{dyn} \Box d_1$
  
  In this case, we use again Definition 3.1.2. Since the only definition of $p$ is $p(x) \leftarrow p(x) \Box x = a$, to avoid a loop, the only choice we have for $J$ is $J = \emptyset$. Moreover, since $FET_\Sigma \models (x \neq a \rightarrow true)^\Sigma$ we have $d_1$ is $x \neq a$. Therefore, we have proved that $P \cup Q \vdash_{dyn} \neg p(x) \leadsto P \cup Q \vdash_{dyn} \Box x \neq a$.

- For $G_2$, we have to prove $P \cup Q \vdash_{dyn} q(x) \leadsto P \cup Q \vdash_{dyn} \Box d_2$
  
  In this case, using Definition 3.1.1, it is almost direct to see that $P \cup Q \vdash_{dyn} q(x) \leadsto P \cup Q \vdash_{dyn} \Box x = a$, hence $d_2$ is $x = a$.

and, finally, we have to prove that $FET_\Sigma \models (true \rightarrow d_1 \vee d_2)^\Sigma$, but this is obvious, since $d_1 \vee d_2$ is $x \neq a \vee x = a$.

As said above, our semantics is not implementable in a straightforward way. In particular, it may be not directly obvious how to find the sets $J$ that are needed in case 3.1.2. Some ideas to implement this case can be found in [9, 10, 35].

3.2. Model-theoretic semantics

In this section, we introduce a class of Beth structures, called ordered structures, following the intuitions described in Section 2.3. In our case, worlds are
pairs \((P,L)\), where \(P\) is a \(\Sigma\)-program and \(L\) is a set of constrained \(\Sigma\)-atoms. The structure associated to a world is also represented (as a variation of Herbrand structures) as a set of constrained \(\Sigma\)-atoms. Moreover, given worlds \(v = (P,L)\) and \(w = (P',L')\), we consider that \(v \preceq w\) if \(P = P'\) and \(L \subseteq L'\). The intuition, in this case, is that the information associated to a world consists of:

- A set of clauses that we may assume to hold (i.e. the program \(P\)).
- A set of facts that we know that do not hold. This set is represented by \(L\).
- A set of facts that we know that do hold. This set is represented by the interpretation of the given world.

Moreover, worlds can be seen as computation stages, where additional computation provides additional knowledge. In this sense, we consider that \(v \preceq w\) if \(w\) includes some more (negative) information than \(v\), when assuming the same program \(P\).

**Example 3.2.** To illustrate the ideas above, let us consider a signature with \(p,q,r,\) and \(s\) as \((0\text{-ary})\) predicate symbols. An ordered structure for this signature may include, for instance, the following worlds and associated interpretations:

\[
\begin{align*}
&\emptyset, \emptyset \quad I(\emptyset, \emptyset) = \emptyset \\
&\emptyset, \{p,q\} \quad I(\emptyset, \{p,q\}) = \{r\} \\
&\emptyset, \{s\} \quad I(\emptyset, \{s\}) = \{p,r\} \\
&\{p \leftarrow \neg q\}, \emptyset \quad I(\{p \leftarrow \neg q\}, \emptyset) = \emptyset \\
&\{p \leftarrow \neg q\}, \{q\} \quad I(\{p \leftarrow \neg q\}, \{q\}) = \{p,r,s\}
\end{align*}
\]

In this structure, the world \((\emptyset, \{p,q\})\) and its associated interpretation \(\{r\}\) represent that, at a certain stage, we may know that \(r\) is true but \(p\) and \(q\) are not. The fact that the program in this world is empty means that we have this knowledge without assuming any clauses. Similarly, the world \((\emptyset, \{s\})\) and its interpretation \(\{p,r\}\) represent that, at a different stage, we may know that \(p\) and \(r\) are true but \(s\) is not. It may be noticed that the worlds \((\emptyset, \{p,q\})\) and \((\emptyset, \{s\})\) are incomparable, with respect to \(\preceq\), meaning that they represent stages following different computation threads.

Then, the world \((\{p \leftarrow \neg q\}, \{q\})\) and its interpretation \(\{p,r,s\}\), represent that, assuming that the clause \(p \leftarrow \neg q\) holds, we may know that \(p\), \(r\) and \(s\) are true while \(q\) is false.

For technical reasons, when defining the notion of model of a given program, we will consider only the structures that, for every program \(P\), the world \((P, \emptyset)\) is present in every structure. We call these structures program-complete ordered structures. Moreover, we assume that the sets of constrained atoms that are included in worlds or in their interpretation are closed under some basic properties. In what follows we denote the set of all constrained atoms by \(\mathcal{L}_\Sigma(X)\).

**Definition 3.3.** A set \(L \subseteq \mathcal{L}_\Sigma(X)\) is \(R\)-closed if it satisfies the following properties:
1. For every $\Sigma$-constraint $c$ such that $FET_\Sigma \models c^3$, we have $\text{true} \circ c \in L$.

2. (Closure under renaming) If $p(\overline{x}) \circ c(\overline{x}) \in L$ then for all renaming of variables $\rho$, $p(\rho(\overline{x})) \circ c(\rho(\overline{x})) \in L$.

3. (Closure under disjunction) If $p \circ c \in L$ and $p \circ d \in L$ then $p \circ c \circ d \in L$.

4. (Closure under general constraints) If $p \circ d \in L$ and $FET_\Sigma \models (c \rightarrow d)^\forall$ then $p \circ c \in L$.

Moreover, given a set $L \subseteq L_\Sigma(X)$, we denote by $\text{Clos}_R(L)$ the smallest R-closed set including $L$.

**Definition 3.4.** Let $\Sigma = (PS_\Sigma, FS_\Sigma)$ be a signature.

1. A $\Sigma$-world $w$ is a pair $(P_w, L_w)$ where $P_w$ is a $\Sigma$-program up to renaming and $L_w \subseteq L_\Sigma(X)$ is R-closed. The set of all the $\Sigma$-worlds is denoted $W_\Sigma$.

2. An ordered $\Sigma$-structure, short ordered structure, is a triple $B = (W, \preceq, I)$, where $W \subseteq W_\Sigma$ and
   
   (a) $\preceq$ is a partial order on $W$, such that for all $v, w \in W : v \preceq w$ if, and only if, $P_v = P_w$ and $L_v \subseteq L_w$. The strict order associated to $\preceq$ is denoted $\prec$.

   (b) The interpretation function $I : W \rightarrow 2^{C_\Sigma(X)}$ satisfies the following properties:
      
      i. For every $v \in W$, $I(v)$ is R-closed.
      
      ii. (Monotonicity) For all $v, w \in W$, if $v \preceq w$ then $I(v) \subseteq I(w)$

3. A program-complete ordered $\Sigma$-structure, short PC ordered structure or PC structure, is an ordered structure such that for every $\Sigma$-program $P$, $(P, \emptyset) \in W$.

The collection of all PC ordered $\Sigma$-structures is denoted $\text{Struct}(\Sigma)$.

Given a program $P$ and a PC ordered structure $B$, $B(P) = (W(P), \preceq, I(P))$ is the ordered structure associated to $P$ occurring in $B$, where $W(P) = \{ w \in W \mid (P, \emptyset) \preceq w \}$ and $I(P) = \{ I(w) \mid w \in W(P) \}$.

Notice that according to this definition, the partial order on $W$, $\preceq$, is uniquely determined by the given set of worlds. As a consequence, in the rest of the paper, when defining a given structure, we will omit the definition of $\preceq$.

For simplicity, we assume the following notational conventions: $\ell \circ c \in I(w)$, $L_w$ means $a \circ c \in I(w)$ if $\ell = a$, and $a \circ c \in L_w$ if $\ell = \neg a$. Conversely, we may write $\neg \ell \circ c \in (I(w), L_w)$ to denote $a \circ c \in L_w$ if $\ell = a$, and $a \circ c \in I(w)$ if $\ell = \neg a$.

Let us now see what do forcing and satisfaction mean in this context. Let $P$ be a program, and $(\emptyset, L)$ a world in a model of $P$. Our intuition is that the literals in $L$ represent some negative consequences of $P$\(^1\) and the literals in the interpretation of $(\emptyset, L)$ represent some positive consequences of $P$. Similarly, the literals in $L$ for a world $(P', L)$ represent negative consequences of $P \cup P'$ (or, equivalently, negative consequences of $P$, under the additional assumptions

---

\(^1\)That is, the consequences which are supported by $P$ at a given computation stage.
in $P'$) and the literals in the interpretation of $(P', L)$ represent positive consequences of $P \cup P'$. As a consequence, forcing is defined like for Beth models: an atomic formula is forced in a world if we know that it will hold in the future, after some possible additional computation. More precisely, a negative literal $\neg a \in c$ is forced in a world $w$ if $a \in c \in L$, for every world $(Q, L)$ in a bar for $w$, and a positive literal $a \in c$ is forced in a world $w$ if $a \in c \in I(v)$, for every world $v$ in a bar for $w$. In the case of non-atomic formulas, the definition provided is the most obvious extension.

**Definition 3.5.** *(Bar)*

1. Let $B = (W, \preceq, I)$ be a PC ordered structure. We say that $B \subseteq W$ is a bar for a world $v \in W$ if for each $\preceq$-increasing chain of worlds $v_0 \preceq v_1 \preceq \ldots$ in $W$ such that $v_0 = v$, there exists $k \geq 0$ such that $v_k \in B$. The bar $B$ is strict if for all worlds $v, w \in B$, $v \not\preceq w$ and $w \not\preceq v$.
2. Let $B_1 = (W_1, \preceq, I_1)$ and $B_2 = (W_2, \preceq, I_2)$ be PC ordered $\Sigma$-structures, $P$ be a $\Sigma$-program, and let $B_1 \subseteq W_1$ and $B_2 \subseteq W_2$ be strict bars for $(P, \emptyset)$. $B \equiv B_2$ if $B_1 = B_2$ and for every $w \in B_1 : I_1(w) = I_2(w)$.

**Definition 3.6.** *(Forcing)* Let $B = (W, \preceq, I)$ be a PC ordered structure. Forcing on $B$, denoted $\vDash$, is inductively defined for every world $v \in W$ as follows:

1. $v, B \vDash \ell \in c$, if there exists a bar $B \subseteq W$ with respect to $v$ such that for all $w \in B : \ell \in c \in (I_B(w), L_w)$.
2. $v, B \vDash \overline{G}_1 \overline{G}_2 \overline{c}$, if $v, B \vDash \overline{G}_1 \overline{c}$, $v, B \vDash \overline{G}_2 \overline{c}$.
3. $v, B \vDash P \supset \overline{G} \overline{c} \overline{c}$, if $(P, \emptyset, \emptyset), B \vDash \overline{G} \overline{c}$.
4. $v, B \vDash p(\overline{x}) \leftarrow \overline{G} \overline{c} \overline{d}$ if for all $w : v \preceq w$ if $w, B \vDash \overline{G} \overline{d}$ then $w, B \vDash p(\overline{x}) \overline{c} \overline{d}$.

We could define the class of models defined by $P$ just as the class of all the PC ordered structures such that the clauses in $P$ are forced by the world $(\emptyset, \emptyset)$. However, this is not satisfactory for our purposes. Many models in that class would not agree with the computational interpretation of our models discussed above, if we impose no condition on the negative information included in worlds. In particular, the atoms in $L$, for a world $(P', L)$, should be supported by the $\Sigma$-program $P \cup P'$ and by the knowledge included in previous worlds. As a consequence, our notion of model for a program $P$ is based on two conditions. The first one is that the program $P$ is forced by all worlds, which means that the interpretation of every world $(P', L)$ should satisfy all the consequences that could be computed from the clauses in $P$ and in $P'$ and from the negative information in $L$. The second condition states that the negative information, $L$, in a world $(P', L)$ must be supported by the clauses in $P$ and in $P'$ and by the information included in previous worlds.

In order to formalize these intuitions we define a notion of local forcing, which can be seen as a kind of local satisfaction on a given world. There are two key ideas in this definition. The first one is to consider that a positive literal $\ell$ is locally forced in a world $w$ if $\ell$ is in the interpretation of $w$, and a negative literal $\ell$ is locally forced in $w = (P, L)$ if $\ell$ is in $L$. The second idea is to consider that in order to see if a formula $P' \supset \ell$ is locally forced in a world $w = (P, L)$
we have to take into account two possible situations. If \( P' \subseteq P \), this means that \( P' \) does not add any new knowledge to the given world, so we have to check if \( \ell \) is locally forced in the \((P, L)\). Conversely, if \( P' \not\subseteq P \), this means that we have to check if \( \ell \) is forced in a world including this new knowledge. In particular, in the world \((P \cup P', \emptyset)\).

**Definition 3.7** (Local forcing). The local forcing relation, \( \models_l \), on a \( \Sigma \)-structure \( B = (W, \leq, I) \) is inductively defined for every world as follows. Let \( v \in W \), then:

1. \( v, B \models_l \ell \circ \circ c \iff \ell \circ \circ c \in (I_B(v), L_v) \).
2. \( v, B \models_l P^1 \supset \ldots \supset P^n \supset \circ \circ c \) if one of the following conditions holds:
   (a) \( P^n = \emptyset \) and \( (P_v \cup P^1 \cup \ldots \cup P^n, \emptyset), B \models_l \circ \circ c \).
   (b) \( P^1 \cup \ldots \cup P^n \subseteq P_v \) and \( v, B \models_l \ell \circ \circ c \).
3. \( v, B \models_l \overline{G}_1, \overline{G}_2 \circ \circ c \) if \( v, B \models_l \overline{G}_1 \circ \circ c \) and \( v, B \models_l \overline{G}_2 \circ \circ c \).
4. \( v, B \models_l p(\tau) \leftarrow \overline{G}_\circ \circ d \) if \( \forall w, v \preceq w \) if \( w, B \models_l \overline{G}_\circ \circ d \) then \( w, B \models_l p(\tau) \circ \circ d \).

Obviously, the relation \( \models_l \) is included in \( \models \), i.e. if \( v, B \models_l \overline{G}_\circ \circ c \) then \( v, B \models \overline{G}_\circ \circ c \).

Next, we define the notion of supported constraint atom with respect to a structure \( B \), a set of worlds \( W \subseteq W_B \), and a program \( P \), meaning that the atom can be safely considered false in the context of the information provided by \( P \) and \( B \), especially, at the worlds in \( W \).

**Definition 3.8** (Supported constrained atoms). A constraint atom \( p \circ \circ c \) is supported with respect to a \( \Sigma \)-structure \( B \), a set of worlds \( W \subseteq W_B \), and a \( \Sigma \)-program \( P \) if for every clause \( (p(\tau) \leftarrow G_1, \ldots, G_m \circ \circ d) \in P \), there exist a subset \( J \subseteq \{1, \ldots, m\} \) and satisfiable constraints \( \{d_j\}_{j \in J} \) such that \( \text{FET}_\Sigma |\models (c \rightarrow -d \lor \bigvee_{j \in J} d_j)^j \) and for all \( j \in J \) there exists \( v \in W \) satisfying that \( v, B \models_l Q_1 \supset \ldots \supset Q_k \supset -\ell \circ \circ d_j \), assuming that \( G_j = Q_1 \supset \ldots \supset Q_k \supset \ell \), for \( 0 \leq k \).

Finally, we can define the notion of model of a program.

**Definition 3.9** (Supported Models). \( B \in \text{Struct}(\Sigma) \) is a model of \( P \), written \( B \models_d P \) if the following two conditions hold:

1. For each \( w \in W_B \), for each clause \( p(\tau) \leftarrow \overline{G}_\circ \circ d \in P \cup P_w \), there exists a \( \Sigma \)-constraint \( c \), \( \text{FET}_\Sigma |\models (c \rightarrow -d)^j \), such that \( w, B \models c(p(\tau) \leftarrow \overline{G}_\circ \circ d) \).
2. Supported worlds: For every world \( w \in W_B \), \( w \) is supported, meaning that every constrained atom \( p \circ \circ c \in L_w \) is supported with respect to \( B \), the set of worlds \( W = \{v \in W_B \mid v \prec w\} \), and the program \( P \cup P_w \).

For every program \( P \), we define the class of its models as:

\[
\text{Mod}(P) = \{ B \mid B \models_d P \}
\]

The following example aims to illustrate the previous definition.
Example 3.3. Let us consider the program $P$ given in Example 3.2. That is, $P = \{(p \leftarrow \{s\} \cup \{r\}); (s \leftarrow p)\}$. A model of $P$ could include any of the ordered structures $B_1$ or $B_2$ described below.

$$B_1 = \begin{cases} 
(\emptyset, \{p, q, s\}) & \{r\} \\
(\emptyset, \{p, q\}) & \{r\} \\
(\emptyset, \emptyset) & \{\{p \leftarrow \lnot q\}, r\} \\
(\emptyset, \emptyset) & \{\{p \leftarrow \lnot q\}, s\}
\end{cases}$$

$$B_2 = \begin{cases} 
(\emptyset, \emptyset) & \{p, s, r\} \\
(\emptyset, \emptyset) & \{p, s, r\} \\
(\emptyset, \emptyset) & \{\{p \leftarrow \lnot q\}, \emptyset\}
\end{cases}$$

When representing ordered structures graphically, $B_1$ and $B_2$ must be read bottom-up. In both of them the increasing chains correspond to the substructures $B_1(\emptyset)$ and $B_1(\{p \leftarrow \lnot q\})$, and $B_2(\emptyset)$ and $B_2(\{p \leftarrow \lnot q\})$, respectively. Additionally, the sets at the right-hand side of the worlds denote their interpretations. It is easy to see that $B_1$ and $B_2$ are models of $P^2$. For instance, considering $B_1$, one can see that the world $(\emptyset, \{p, q, s\})$ is supported by the world $(\emptyset, \{p, q\})$ and this later one is (trivially) supported by the world $(\emptyset, \emptyset)$. Additionally, each world locally forces each clause in $P$.

3.3. Least fixpoint semantics

In this section, we define an immediate consequence operator $T_P$ that can be used to build (bottom-up) its least fixpoint, which is shown to be a model of the given program $P$. In particular, this fixpoint will be shown to be the least model in $\text{Mod}(P)$, with respect to an ordering that will be defined in the following section. Moreover, the operational semantics defined above will be shown to be sound and complete with respect to that model.

In particular, $T_P$ is a kind of embedding where, for every world $w = (P, L)$ in a structure $A$, there is a corresponding world $w' = (P, L')$, with $L \subseteq L'$ in $T_P(A)$, and where the interpretation in $T_P(A)$ of $w'$ also includes the interpretation $w$ in $A$ of $w$. More precisely, the worlds and their interpretation in $T_P(A)$ include the negative information that is supported by the existing worlds and their interpretation in $A$.

To define $T_P$, we use two auxiliary mappings, $\text{Neg}_P^A$ and $\text{Pos}_P^A$. The first one, $\text{Neg}_P^A$, given a world $w$ in $A$, yields a $\Sigma$-world $v$, not necessarily different from $w$, including the negative information supported (in the sense of Definition 3.9) by $w$.

Definition 3.10. Let $A \in \text{Struct}(\Sigma)$ and let $P$ be a $\Sigma$-program. The mapping $\text{Neg}_P^A : W_A \rightarrow W_\Sigma$ is defined as follows. For every world $w = (P', L) \in W_A$,

---

To be precise $B_1$ and $B_2$ are not really models of $P$, since they are not program-complete. They would be models of $P$ if we would add, at least, all the worlds $(Q, \emptyset)$ and their corresponding interpretations, for every program $Q$.  

17
\( \text{Neg}_A^P(w) = (P', \text{Clos}_R(L')) \), where \( L' \) is the set of all constrained atoms \( p(\bar{x}) \) that are supported with respect to \( A \), the set \{ \( w \) \} and the program \( P \cup P_w \).

\( \text{Pos}_A^P \) is a mapping from the set of worlds in \( \mathcal{A} \) into the class of all sets of constrained \( \Sigma \)-atoms. In particular, given a world \( (P', L) \), \( \text{Pos}_A^P(P', L) \) is the set of all constrained atoms that are a direct consequence of \( P \cup P' \), of the positive information included in the interpretation of \( (P', L) \) in \( \mathcal{A} \), and of the negative information in \( L \).

**Definition 3.11.** Let \( \mathcal{A} \in \text{Struct}(\Sigma) \) and let \( P \) be a \( \Sigma \)-program. The mapping \( \text{Pos}_A^P : W_\mathcal{A} \rightarrow 2^{L_\mathcal{X}(\Sigma)} \) is defined as follows. For every world \( w = (P', L) \in W_\mathcal{A} \), \( \text{Pos}_A^P((w)) = \text{Clos}_R(A) \), where \( A \) is the set of all constrained atoms that are supported with respect to \( w \). Then, for every world \( w \) its interpretation in \( \text{Pos}_A^P((w)) \) is a PC ordered structure for every \( \mathcal{A} \). As we will see below, the bottom \( \Sigma \)-structure, \( \bot \), is just a structure where, for every program \( P' \), we just have a world \( (P', \emptyset) \) whose interpretation is the empty set of atoms. In the rest of the example, we will just show how the operator acts on the worlds associated to the empty program, \( P_1 = \emptyset \) and the program \( P_2 = \{(p \leftarrow q)\} \). To start, let

\[ \bot = \{ (\emptyset, \emptyset) \} \]

1. **First iteration, \( T_P(\bot) \)**
   a. Applying \( \text{Neg}_P \).

It may be noted that, in the above definitions, we make sure that the results of \( \text{Neg}_A^P \) and \( \text{Pos}_A^P \) are closed by the properties stated in Definition 3.3, so that \( T_P(\mathcal{A}) \) is a PC ordered structure for every \( \mathcal{A} \) in the following definition.

**Definition 3.12.** Let \( P \) be a \( \Sigma \)-program. The mapping \( T_P : \text{Struct}(\Sigma) \rightarrow \text{Struct}(\Sigma) \) is defined as follows. For each \( \mathcal{A} = (W_\mathcal{A}, \preceq_\mathcal{A}, \mathcal{I}_\mathcal{A}) \in \text{Struct}(\Sigma) \), \( T_P(\mathcal{A}) \) is the PC ordered structure \( \mathcal{A}' = (W_\mathcal{A}', \preceq_\mathcal{A}', \mathcal{I}_\mathcal{A}') \) where:

1. \( W_\mathcal{A}' = \{(P', \emptyset) \mid P' \text{ is a } \Sigma \text{-program}\} \cup \{\text{Neg}_A^P(w) \mid w \in W_\mathcal{A}\} \)
2. For each \( w \in W_\mathcal{A}' \), \( \mathcal{I}_\mathcal{A}'(w) = \text{Clos}_R(\cup_{v \in W_{\mathcal{A}'}, v \preceq w} \text{Pos}_A^P(v)) \)

The set of worlds in \( T_P(\mathcal{A}) \) includes all the worlds \( (P', \emptyset) \) for each \( \Sigma \)-program \( P' \) (since \( T_P(\mathcal{A}) \) must be program complete) plus all the words computed using \( \text{Neg}_A^P \), i.e. all the worlds that include the additional negative information that is supported by \( \mathcal{A} \). Then, for every world \( w \) its interpretation in \( T_P(\mathcal{A}) \) includes all the positive information that is a direct consequence of the information in all the worlds \( v \) with \( v \preceq w \).

In order to illustrate the way in which \( T_P \) works and the order \( \preceq_F \), let us consider the following example:

**Example 3.4.** Let us see the construction of the least fix-point of the program \( P \) given in Example 3.3. That is, \( P = \{(r \leftarrow \{(p \leftarrow q) \supset s\}; \{(s \leftarrow p)\}\} \).

As we will see below, the bottom \( \Sigma \)-structure, \( \bot \), is just a structure where, for every program \( P' \), we just have a world \( (P', \emptyset) \) whose interpretation is the empty set of atoms. In the rest of the example, we will just show how the operator acts on the worlds associated to the empty program, \( P_1 = \emptyset \) and the program \( P_2 = \{(p \leftarrow q)\} \). To start, let

\[ \bot = \{ (\emptyset, \emptyset) \} \]

1. **First iteration, \( T_P(\bot) \)**
   a. Applying \( \text{Neg}_P \).
ii. Neg$^P_P((p \leftarrow \lnot q), \emptyset)) = ((p \leftarrow q), \emptyset))$. On the one hand, Def$^P_P((p \leftarrow q), \emptyset), q \in L'_w$. On the other hand, using similar arguments as in the previous case, we can easily see that $p, r$ and $s$ are not supported.

Therefore, $W_{T_P(\bot)}$ contains $\{(\emptyset, \emptyset), (\{p \leftarrow q\}, \emptyset) \} \cup \{(\emptyset, \{p, q\}, (\{p \leftarrow \lnot q\}, \{q\})\}$.

(b) Applying $Pos^P_P$:

i. $Pos^P_P((\emptyset, \emptyset)) = \emptyset$. The reason is that the only clauses in $P$ are $s \leftarrow p$ and $r \leftarrow \{p \leftarrow q\} \supset s$. In the former case, it is obvious that $(\emptyset, \emptyset), \models \top \models p$. In the latter case, we need to check if $(\emptyset, \emptyset), \models \top \models \{p \leftarrow q\}, \emptyset, \models r$. However, it should be clear that $(\{p \leftarrow \lnot q\}, \emptyset), \models \top \models s$.

ii. Similarly, it is easy to see that $Pos^P_P((\{p \leftarrow \lnot q\}, \emptyset)) = \emptyset$.

Hence, by definition of $T_P$, we have:

$$T_P(\bot) = \begin{cases} \{(\emptyset, \emptyset)\} & (\{p \leftarrow \lnot q\}, \emptyset) \} \\ \{(\emptyset, \emptyset)\} & (\{p \leftarrow \lnot q\}, \emptyset) \} \end{cases}$$

2. Second iteration, $T^2_P(\bot)$

(a) Applying Neg$^{T^2_P(\bot)}_P$:

i. Using the same arguments as above, Neg$^{T^2_P(\bot)}_P((\emptyset, \emptyset)) = (\emptyset, \{p, q\})$ and Neg$^{T^2_P(\bot)}_P((\{p \leftarrow \lnot q\}, \emptyset)) = ((p \leftarrow q), \{q\})$.

ii. Neg$^{T^2_P(\bot)}_P((\emptyset, \{p, q\})) = (\emptyset, \{p, q, s\})$. On the one hand, $p, q$ and $s$ are supported, since there are no clauses defining $p$ and $q$ and, for the only clause defining $s$, $s \leftarrow p$, we have that $(\emptyset, \{p, q\})), T_P(\bot) \models \top \models p$. On the other hand, as above, it is easy to see that $r$ is not supported.

iii. Finally, using similar arguments, it is easy to see that Neg$^{T^2_P(\bot)}_P(((p \leftarrow \lnot q), \{q\})) = (\{p \leftarrow \lnot q\}, \{q\})$.

Therefore, $W_{T^2_P(\bot)}$ contains $\{(\emptyset, \emptyset), (\{p \leftarrow q\}, \emptyset) \} \cup (\emptyset, \{p, q\}), (\emptyset, \{p, q, s\}), (\{p \leftarrow \lnot q\}, \{q\})$.

(b) Applying Pos$^{T^2_P(\bot)}_P$:

i. Using similar arguments as in the previous iteration, it is easy to see that, for all worlds $w$ except $(\{p \leftarrow q\}, \{q\})$, Pos$^{T^2_P(\bot)}_P(w)$ is empty.
3. **Third iteration, \( T^3_p(\bot) \)**

   (a) Using similar reasoning as above, it is easy that \( \text{Neg}_{T^3_p(\bot)} \) does not add any new world.

   (b) Applying \( \text{Pos}_{T^3_p(\bot)} \).

   i. Using similar arguments as in the previous iteration, it is easy to see that, for all worlds \( w \) except \( \{p \leftarrow \neg q\}, \{q\} \), \( \text{Pos}_{T^3_p(\bot)}(w) \) is empty.

   ii. \( \text{Pos}_{T^3_p(\bot)}((\{p \leftarrow \neg q\}, \{q\})) = \{p, s\} \). We have seen the case of \( p \) in the previous iteration. In the case of \( s \), we have the clause \( (s \leftarrow p) \) in \( P \) and \( \{p \leftarrow \neg q\}, \{q\} \), \( T^3_p(\bot) \) \( \models \) \( p \). In addition, it is easy to see that no other atom is in \( \text{Pos}_{T^3_p(\bot)}((\{p \leftarrow \neg q\}, \{q\})) \).

   Hence, we have:

   \[
   T^3_p(\bot) = T_p(T^3_p(\bot)) = \begin{cases} 
   (\emptyset, \{p, q, s\}) \emptyset \\
   (\emptyset, \{p, q\}) \emptyset \\
   (\emptyset, \emptyset) \emptyset 
   \end{cases} 
   \]

4. **Fourth iteration, \( T^4_p(\bot) \)**

   (a) Using similar reasoning as above, it is easy that \( \text{Neg}_{T^4_p(\bot)} \) does not add any new world.

   (b) Applying \( \text{Pos}_{T^4_p(\bot)} \).

   i. \( \text{Pos}_{T^4_p(\bot)}((\emptyset, \emptyset)) = \{r\} \). The reason is that we have the clause \( (r \leftarrow \{p \leftarrow \neg q\}) \) in \( P \) and we have \( \{p \leftarrow \neg q\}, \emptyset \), \( T^3_p(\bot) \) \( \models \) \( s \), which means that \( (\emptyset, \emptyset), T^3_p(\bot) \) \( \models \) \( \{p \leftarrow \neg q\} \) \( \supset \) \( s \).

   ii. For similar reasons, \( \text{Pos}_{T^4_p(\bot)}((\emptyset, \{p, q\})) = \{r\} \).

   iii. And also, \( \text{Pos}_{T^4_p(\bot)}((\emptyset, \{p, q, s\})) = \{r\} \).

   iv. As in the above iteration, \( \text{Pos}_{T^4_p(\bot)}((\{p \leftarrow \neg q\}, \emptyset)) = \emptyset \).
v. Finally, $\text{Pos}^T_\Sigma_p(\{(p \leftarrow \neg q), \{q\}\}) = \{p, r, s\}$. In particular, in previous iterations we have already seen why $p$ and $s$ are in this set. In the case of $r$, we have the clause $(r \leftarrow \{(p \leftarrow \neg q) \cup s\}$ in $P$, therefore we have to prove that $\{(p \leftarrow \neg q), \{q\}, T^\Sigma_\Sigma_p(\bot)\}$ \hspace{2mm} $\vdash_1 \{(p \leftarrow \neg q) \cup s\}$. By Definition 3.7.2, and considering that $\{p \leftarrow \neg q\} \subseteq P \cup \{p \leftarrow \neg q\}$, this means showing $((p \leftarrow \neg q), \{q\}, T^\Sigma_\Sigma_p(\bot)) \vdash_1 \{(\neg q) \cup s\}$, which trivially holds.

Hence, we have:

$$T^\Sigma_\Sigma_p(\bot) = T^\Sigma_\Sigma_p(T^\Sigma_\Sigma_p(\bot)) = \left\{ \begin{array}{ccc} (\emptyset, \{p, q, s\}) \{r\} & | & (\{p \leftarrow \neg q\}, \{q\}) \{p, s, r\} \\ (\emptyset, \{p, q\}) \{r\} & | & (p \leftarrow \neg q), \emptyset \{r\} \end{array} \right\}$$

Finally, it is easy to see that $T^\Sigma_\Sigma_p(\bot) = T^\Sigma_\Sigma_p(\bot)$, which means that this is the least fix-point.

In order to prove that $T_\Sigma_p$ has a least fixpoint, we can define an order relation on PC ordered $\Sigma$-structures according to the amount of information they contain. In particular, given two structures $\mathcal{A}$ and $\mathcal{B}$, we consider that $\mathcal{A} \preceq_\Sigma \mathcal{B}$ if the positive and negative information in $\mathcal{A}$ is included in the positive and negative information in $\mathcal{B}$. However, defining this orderings for arbitrary PC ordered structures is a bit involved and not really useful for our main aim. Instead, we will just define the order relation for linear ordered structures, which are structures where, for every program $P$, the worlds over $P$ form an ascending sequence $(P, \emptyset) \preceq (P, L_1) \preceq \ldots \preceq (P, L_k) \preceq \ldots$. This is enough for our purposes since, for every $n \leq \omega$, $T^\Sigma_p(\bot)$ is a linear ordered structure.

**Definition 3.13.** A linear ordered $\Sigma$-structure $\mathcal{A}$ is an ordered structure where, for every $\Sigma$-program $P$, the set $\{(P, L) \mid (P, L) \in W^\mathcal{A}\}$ is an ascending sequence $(P, \emptyset) \preceq (P, L_1) \preceq \ldots \preceq (P, L_k) \preceq \ldots \preceq (P, L_n)$, where $n \leq \omega$.

Given linear ordered $\Sigma$-structures $\mathcal{A}$ and $\mathcal{A}'$, $\mathcal{A} \preceq_\Sigma \mathcal{A}'$ if, for every $\Sigma$-program $P$ and the associated ascending sequences $(P, \emptyset) \preceq (P, L_1) \preceq \ldots \preceq (P, L_k) \preceq \ldots \preceq (P, L_n)$ and $(P, \emptyset) \preceq (P, L'_1) \preceq \ldots \preceq (P, L'_k) \preceq \ldots \preceq (P, L'_m)$ in $\mathcal{A}$ and $\mathcal{A}'$, respectively, $n \leq m$ and the following two conditions hold:

1. For every $i \leq n$ : $L_i \subseteq L'_i$,
2. For every $i \leq n$ : $I^\mathcal{A}_i((P, L_i)) \sqsubseteq I^\mathcal{A}'((P, L'_i))$

where $L_0 = L'_0 = \emptyset$

It is not difficult to see that $\preceq_\Sigma$ is a complete partial order over the class of all linear PC structures. In particular, the least upper bound for an ascending chain $\mathcal{A}_1 \preceq_\Sigma \mathcal{A}_2 \preceq_\Sigma \ldots$ is a structure $\mathcal{A}_\cup = (W_\cup, \preceq, I_\cup)$, where $W_\cup$ is the union of the set of worlds of the structures in the chain, and, for every world $w$, $I_\cup(w)$ is the union of the interpretations of $w$ in the structures that include $w$.
Theorem 3.1. The relation $\preceq_F$ is a complete partial order on linear PC structures.

We can also show that $T_P$ is monotonic and continuous and, therefore, it has a least fixpoint at the $\omega$ iteration over the $\bot$ structure. The proof is done using the fact that the mappings $Pos^A_P$ are monotonic and continuous.

Theorem 3.2. For any $\Sigma$-program $P$, $T_P$, when restricted to the class of linear PC structures, is a monotonic and continuous operator with respect to $\preceq_F$ so, it has a least fix-point $T_P \uparrow \omega$.

Finally, we can see that, as expected, the least fix-point of the operator $T_P$ is a model of the $\Sigma$ program $P$.

Proposition 3.1. Given a $\Sigma$ program $P$, $T_P \uparrow \omega \in Mod(P)$.

3.4. Least model semantics

In this section we prove that the least fixpoint of the immediate consequence operator $T_P \uparrow \omega$ is the least model in $Mod(P)$ with respect to a proper notion of ordering. The key issue here is to define an ordering relation in $Mod(P)$, which we will denote by $\sqsubseteq$, such that it adequately captures the intuition that the “best model” is the least one. The definition of this ordering is based, first, on the definition of an ordering between ordered structures associated to a given program $P$. Then this ordering is extended to compare $\Sigma$-structures by comparing the ordered structures included.

We may notice that, in an ordered structure associated to a program $P$, (if this structure is part of a model of a program $P'$) the negative information associated to a given world will contain, at most, the negative information supported by the worlds below. Similarly, the positive information associated to a given world will contain, at least, all the consequences that can be computed from the clauses in $P$ and in $P'$ and the negative information in the world. In this sense, one may consider that the best ordered structure is one in which the negative and positive information associated to each world is, respectively, the maximum and the minimum amount of possible information.

This means that the ordering between ordered structures should be based on an extension of the, so-called, standard ordering of 3-valued structures [32]. In this ordering, the least models of a program $P$ minimize and maximize the positive and negative information, respectively.

However, given two ordered structures $B_1(P)$ and $B_2(P)$, we should not try to compare pairwise all the worlds in one structure with the associated worlds in the other. For instance, let us suppose that $P$ consists of the clause $r \leftarrow \neg q$ and $P'$ is empty. If $q$ is not included in the interpretation of the world $(P, \emptyset)$ in $B_1(P)$ then the world above may be $(P, \{q\})$ and its interpretation would include $r$. But, if $q$ is included in the interpretation of $(P, \emptyset)$ in $B_2(P)$, then the world above can be $(P, \{r\})$. Obviously, $B_1(P)$ should be considered better than $B_2(P)$. We can define an ordering meeting these intuitions as some kind of lexicographic extension of the standard ordering.
The fact that ordered structures may be not linear poses some small additional difficulty: two structures may be incomparable but, at the same time, be defined over the same set of worlds. Nevertheless, with the intuition discussed above, to compare $B_1(P)$ and $B_2(P)$ we proceed as follows. First, we look for two bars $B_1$ and $B_2$ in $B_1(P)$ and $B_2(P)$, respectively, in each structure. Then, if all the worlds and interpretations in the segment of the substructure $B_1(P)$ below $B_1$ are subsumed by the segment of the substructure $B_2(P)$ below $B_2$ and, all the worlds and interpretations in $B_1$ are smaller (with respect to the standard ordering) than all the worlds and interpretations in $B_2$, then we consider $B_1(P)$ smaller than $B_2(P)$.

The following definitions capture these intuitions:

**Definition 3.14.**

1. Let $B_1 = \langle W_1, \preceq, I_1 \rangle$ and $B_2 = \langle W_2, \preceq, I_2 \rangle$ be $\Sigma$-structures, $P$ a $\Sigma$-program, $B_1(P)$ and $B_2(P)$ the ordered structures associated to $P$, in $B_1$ and $B_2$, respectively. Then, $B_2(P)$ subsumes $B_1(P)$, denoted by $B_1(P) \subseteq B_2(P)$, if, and only if, $W_1(P) = W_2(P)$ and $\forall w \in W_1(P)$, $I_1(P)(w) \subseteq I_2(P)(w)$.

2. Given a $\Sigma$-structure $B = \langle W, \preceq, I \rangle$, a $\Sigma$-program $P$ and a strict bar $B \subseteq W$ with respect to $(P, \emptyset)$, we define $B_1 = \langle W_{B_1}, \preceq, I_{B_1} \rangle$ such that $W_{B_1} = \{ v \in W(P) \mid \exists w \in B \text{ and } v \prec w \}$ and $I_{B_1} = \{ I(w) \mid w \in W_{B_1} \}$.

**Definition 3.15.** Let $B_1$ and $B_2$ be $\Sigma$-structures, $P$ a $\Sigma$-program and $B_1(P)$ and $B_2(P)$ the ordered structures associated to $P$ in $B_1$ and $B_2$, respectively. $B_1(P) \subseteq_s B_2(P)$ if, and only if, one of the following conditions hold:

1. $B_1(P) \subseteq B_2(P)$

2. There exist strict bars $B_i \subseteq W_i(P)$, $i \in \{1, 2\}$ with respect to $(P, \emptyset)$ in $B_1(P)$ and $B_2(P)$, respectively, such that $B_1 \downarrow = B_2 \downarrow$ and, for all $\preceq$-increasing chains $v_0, \ldots, v_k, w_1$ and $v_0, \ldots, v_k, w_2$ in $B_1(P)$ and $B_2(P)$, respectively, where $v_0 = (P, \emptyset)$ and $w_i \in B_i$, $i \in \{1, 2\}$, the following condition holds:

   $$L_{w_2} \subseteq L_{w_1} \text{ or } (L_{w_2} = L_{w_1} \text{ and } I_1(w_1) \subseteq I_2(w_2))$$

The strict order associated to this definition is denoted $\subseteq_s$.

It is just routine to prove that $\subseteq_s$ is a partial order.

**Proposition 3.2.** The relation $\subseteq_s$ over ordered structures associated to a $\Sigma$-program $P$ is a partial order.

**Example 3.5.** Let us considerer the following ordered structures:

\[
B_1 = \begin{cases}
(\emptyset, \{p, q, s\}) \{r\} \\
B_1 \quad \boxed{(\emptyset, \{p, q\}) \{r\}} \quad ((p \leftarrow \neg q), \{q\}) \{p, r, s\} \\
(\emptyset, \emptyset) \emptyset \quad (\{p \leftarrow \neg q\}, \emptyset) \emptyset \quad B_1'
\end{cases}
\]
Definition 3.16. \( \preceq \) is the smallest order relation satisfying that if a clause of the form \( p(\overline{x}) \leftarrow G_1, \ldots, G_n, \overline{cc} \in Q, \ n > 0 \) and \( \exists j: 1 \leq j \leq n, \ G_j = Q_1 \supset \ldots \supset P \supset \ldots \supset Q_k \supset \ell_j, \ k \leq 0 \) then \( P \preceq Q \).

It is easy to see that the relation \( \preceq \) is a partial order. In particular, since we do not allow named program units, antisymmetry property holds.

Definition 3.17. Given \( \mathcal{B}_1 \) and \( \mathcal{B}_2 \) both in \( \text{Struct}(\Sigma) \), \( \mathcal{B}_1 \preceq \mathcal{B}_2 \) if, either \( \mathcal{B}_1 = \mathcal{B}_2 \), or for each chain \( P_1 \preceq \ldots \preceq P_i \preceq \ldots \) of \( \Sigma \)-programs, there exists \( i \in \mathbb{N} \) such that the following conditions hold:

\[
\forall j \in \mathbb{N}, \ j < i: \ \mathcal{B}_1(P_j) = \mathcal{B}_2(P_j) \text{ and } \mathcal{B}_1(P_i) \preceq \mathcal{B}_2(P_i)
\]
As in previous definition, it is easy to see that the relation $\sqsubseteq$ is a partial order.

**Theorem 3.3.** The relation $\sqsubseteq$ over $\text{Struct}(\Sigma)$ is a partial order.

Now we can state the main result of this subsection:

**Theorem 3.4.** For any $\Sigma$-program $P$, $T_P \uparrow_\omega$ is the $\sqsubseteq$-least model in $\text{Mod}(P)$.

3.5. Soundness and completeness

Finally, we can show the soundness and completeness of the operational semantics of a program $P$ with respect to the least fixpoint of the immediate consequence operator $T_P$. To prove soundness, it is convenient to prove the following lemma:

**Lemma 3.1.** Let $P$ be a $\Sigma$-program and $\overline{G} \sqsubseteq c$ a $\Sigma$-goal. For every $\Sigma$-program $Q$, if $P \cup Q \vdash_{\text{dyn}} \overline{G} \sqsubseteq c \Rightarrow_{\text{dyn}} c'$, then $(Q, \emptyset), T_P \uparrow_\omega \models \overline{G} \sqsubseteq c'$.

As an immediate consequence, we have:

**Theorem 3.5** (Soundness of the operational semantics). Let $P$ be a $\Sigma$-program and $\overline{G} \sqsubseteq c$ a $\Sigma$-goal. If $P \vdash_{\text{dyn}} \overline{G} \sqsubseteq c \Rightarrow_{\text{dyn}} c'$, then $(\emptyset, \emptyset), T_P \uparrow_\omega \models \overline{G} \sqsubseteq c'$.

Similarly, to prove completeness, we first show the following lemma, where the notation $P \vdash_{\text{dyn}} p(\overline{x}) \sqsubseteq c \Rightarrow_{\text{dyn}} c_1, \ldots, c_n$ means that $P \vdash_{\text{dyn}} p(\overline{x}) \sqsubseteq c$ can be proved with computed answers $c_1, \ldots, c_n$.

**Lemma 3.2.** Let $P$ be a $\Sigma$-program and $\overline{G} \sqsubseteq c$ a $\Sigma$-goal. For every $\Sigma$-program $Q$ if $(Q, \emptyset), T_P \uparrow_\omega \models \overline{G} \sqsubseteq c$, then $P \cup Q \vdash_{\text{dyn}} \overline{G} \sqsubseteq c \Rightarrow_{\text{dyn}} c_1, \ldots, c_n$ such that $\text{FET}_\Sigma \models (c \rightarrow \bigvee_{i=1}^n c_i)^\forall$.

And, again, as an immediate consequence, we have:

**Theorem 3.6** (Completeness of the operational semantics). Let $P$ be a $\Sigma$-program and $\overline{G} \sqsubseteq c$ a $\Sigma$-goal. If $(\emptyset, \emptyset), T_P \uparrow_\omega \models \overline{G} \sqsubseteq c$, then $P \vdash_{\text{dyn}} \overline{G} \sqsubseteq c \Rightarrow_{\text{dyn}} c_1, \ldots, c_n$ such that $\text{FET}_\Sigma \models (c \rightarrow \bigvee_{i=1}^n c_i)^\forall$.

4. A transformational semantics of static embedded implications of normal logic programs

In this section, we consider the same kind of programs as in Section 3, but interpreting embedded with static scoping. However, we follow an approach that could be considered more pragmatic. Instead of developing a new framework to define a model-theoretic semantics of this class of programs, we show how these programs can be transformed into standard normal programs. Moreover, we prove that this translation is sound and complete with respect to the operational semantics of the extended programs. In addition, it must be pointed out that this transformation is easy to implement, which means that we can easily build this kind of extension on top of a standard logic programming language.
This approach has been used in [28] to deal with positive propositional static programs. Indeed, herein we actually extend that work. This approach has also been used in [3] to translate modal logic programs with embedded implication into Horn programs.

The section is organized as follows. First, we introduce an operational semantics for the class of static normal logic programs with embedded implications. Then, Section 4.2 presents the transformation semantics and, finally, in Section 4.3, we prove the soundness and completeness of the transformation.

4.1. Operational semantics

In this section, we propose an operational semantics which can be seen as a combination of the operational semantics defined in [17] and SLDFA resolution [9].

To provide some intuition, let us first see how static scoping affects the operational semantics of positive programs including embedded implications. The rule for handling implications in this case [17] is shown in Figure 3. As we can see, it is quite similar to the corresponding rule for the dynamic case, which is shown in Figure 2.

\[
\begin{align*}
\frac{P \perp Q \vdash_{st} G}{P \vdash_{st} Q \supset G}
\end{align*}
\]

Figure 3: Static rule (S-rule)

Obviously, to understand the static rule and its difference with respect to the dynamic rule, we have to understand what is the meaning of \( P \perp Q \vdash_{st} G \). The intuition of this notation is that we want to solve \( G \) in a scope consisting of two nested blocks of definitions, \( P \) and \( Q \), where \( Q \) is local to \( P \), i.e. \( P \perp Q \) represents this kind of block inclusion. Now, when we are trying to solve the goal \( G \) in the scope of \( P \perp Q \) we can use any definition that is present in either \( P \) or \( Q \). This is just as in block-structured procedural languages where, for solving a reference in a given scope, we may use any visible definition from any of the blocks which are global to that reference. Let us suppose that we have the clause \(( G \leftarrow G_1, \ldots, G_n) \) in \( P \) and, moreover, that this is the clause that we use when trying to solve \( G \). Then, now we would need to solve the new goals \( G_1, \ldots, G_n \), but we would only be allowed to use clauses from \( P \), since these goals come from a clause in \( P \) and the definitions in \( Q \) are considered to be invisible in \( P \). This kind of inference is formulated (in a restricted version) in Figure 4.

\[
\begin{align*}
\frac{P_1 \ldots | P_i | \vdash_{st} G_1, \ldots, G_n}{P_1 \ldots | P_i | \ldots | P_k \vdash_{st} G} \quad \text{if} \quad (G \leftarrow G_1, \ldots, G_n) \in P_i
\end{align*}
\]

Figure 4: Static resolution-like rule
To show in more detail the difference between dynamic and static operational semantics, let us consider the following simple propositional normal logic programs and the derivations in both approaches:

**Example 4.1.** Let $P = \{(t \leftarrow q)\}$ and $Q = \{(q)\}$.

<table>
<thead>
<tr>
<th>Dynamic</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \vdash_{\text{dyn}} Q \cup p$</td>
<td>$P \vdash_{\text{st}} Q \cup p$</td>
</tr>
<tr>
<td>$P \vdash_{\text{dyn}} Q \cup \neg p$</td>
<td>$P \vdash_{\text{st}} Q \cup \neg p$</td>
</tr>
</tbody>
</table>

As a consequence of how we deal with these nested sets of definitions, we call $P_1|\ldots|P_k$ a stack of programs. In particular, if we apply the rule in Figure 3, we consider that we are pushing $Q$ to the given stack. Conversely, if we apply the rule in Figure 4, we consider that we are popping programs $P_{k+1}|\ldots|P_k$ from the given stack.

**Definition 4.1.** (Stack of programs) Given the programs $P_1, \ldots, P_k$, $k > 0$, a stack of programs $S_k$ is a sequence $\langle\emptyset, P_1, \ldots, P_k\rangle$ written as $P_1|\ldots|P_k$. A stack of programs increases/decreases following a LIFO strategy, assuming that the last added program is $P_k$. The length of $S_k$ is $k$.

For technical reasons we assume that every stack includes the empty program at the bottom.

The semantics defined below is obviously quite close to the one defined in Definition 3.1 for normal logic programs with embedded implications when using dynamic scoping. In this case, the new derivation relation is presented in terms of mutually recursive definitions over sequents of the form $S \vdash_{\text{st}} \overline{\text{G}}_{\text{ccc}}$, where $S$ is a stack of programs and $\overline{\text{G}}_{\text{ccc}}$ is a goal.

**Definition 4.2.** The derivation relation $\vdash_{\text{over sequents}}$ is defined as follows:

1. $P_1|\ldots|P_k \vdash_{\text{st}} \overline{\text{G}}_1, p(x), \overline{\text{G}}_{2\text{ccc}} \vdash P_1|\ldots|P_k \vdash_{\text{st}} \overline{\text{G}}_1, \overline{\text{G}}_{2\text{ccc}}'$ if there exists $i \in \{1, \ldots, k\}$ and a (renamed apart) clause $p(x) \leftarrow \overline{\text{G}}_d \in \text{Def}(P_i, p)$ such that $P_1|\ldots|P_k \vdash_{\text{st}} \overline{\text{G}}_1 \land \overline{\text{G}}_d \vdash\overline{\text{G}}_c \vdash P_i|\ldots|P_k \vdash_{\text{st}} \overline{\text{G}}_{2\text{ccc}}$.
2. $P_1|\ldots|P_k \vdash_{\text{st}} \overline{\text{G}}_1, \neg p(x), \overline{\text{G}}_{2\text{ccc}} \vdash P_1|\ldots|P_k \vdash_{\text{st}} \overline{\text{G}}_1, \overline{\text{G}}_{2\text{ccc}}'$ if for every $i \in \{1, \ldots, k\}$ and every (renamed apart) clause $p(x) \leftarrow G_1, \ldots, G_m \vdash d \in \text{Def}(P_i, p)$ there exists $J \subseteq \{1, \ldots, m\}$, not necessarily unique, such that for all $j \in J$, assuming $G_j = Q_1 \cup \ldots \cup Q_n \cup \ell$ with $0 \leq n$, we have $P_1|\ldots|P_i|Q_{i+1}|\ldots|Q_n \vdash_{\text{st}} \neg \ell \vdash \overline{\text{G}}_c \vdash P_i|\ldots|P_k \vdash_{\text{st}} d_J$, and $\text{FET}_\Sigma \models (c' \rightarrow \neg d \lor \bigvee_{j \in J} d'_j)$.\]

3. $P_1|\ldots|P_{k-1} \vdash_{\text{st}} \overline{\text{G}}_1, P_k \cup G, \overline{\text{G}}_{2\text{ccc}} \vdash P_1|\ldots|P_{k-1} \vdash_{\text{st}} \overline{\text{G}}_1, \overline{\text{G}}_{2\text{ccc}}'$ if $P_1|\ldots|P_k \vdash_{\text{st}} \overline{\text{G}}_{\text{ccc}} \vdash P_1|\ldots|P_k \vdash_{\text{st}} \overline{\text{G}}_{\text{ccc}}'$.\]

Each item in this definition is called a derivation step.

**Definition 4.3.** Let $S$ be a stack of programs and $\overline{\text{G}}_{\text{ccc}}$ a goal. $S \vdash_{\text{st}} \overline{\text{G}}_{\text{ccc}}$ can be statically proved with computed answer $c'$, denoted $S \vdash_{\text{st}} \overline{\text{G}}_{\text{ccc}} \Rightarrow_{\text{st}} c'$, if $S \vdash_{\text{st}} \overline{\text{G}}_{\text{ccc}} \Rightarrow S \vdash_{\text{st}} \overline{\text{G}}_{\text{ccc}}'$, $\text{FET}_\Sigma \models c'^2$, and $\text{FET}_\Sigma \models (c' \rightarrow c)^\forall$.\]
As in Section 3.1, we assume that whenever an expression of the form \( \neg p(\overline{x})c \) occurs in the right-hand side of a sequent, it denotes \( p(\overline{x})c \). Next we give the intuition behind our operational semantics. In particular, we present some examples to show how the operational semantics works.

**Example 4.2.** Let \( P = \{ (p(x) \leftarrow \overline{x} = a) \} \) and \( Q = \{ (p(x) \leftarrow \overline{x} = b) \} \). The derivation \( P \vdash_{st} Q \vdash_{st} \neg p(x) \dashv P \vdash_{st} \overline{x} \neq a \land x \neq b \) is justified because the following subderivations:

- \( P|Q \vdash_{st} \neg p(x) \dashv P|Q \vdash_{st} \overline{x} \neq b \).
- \( P \vdash_{st} \neg p(x) \dashv P \vdash_{st} \overline{x} \neq a \)

The following example adapts the one presented in [9].

**Example 4.3.** Let \( P = \{ (r \leftarrow Q_1 \supset \neg p(x), Q_2 \supset \neg q(x)) \}, Q_1 = \{ (p(x) \leftarrow p(x)); (p(x) \leftarrow \overline{x} = a) \} \) and \( Q_2 = \{ (q(x) \leftarrow q(x)\overline{x} = a); (q(x) \leftarrow \neg s(x)); (s(x) \leftarrow \overline{x} = a) \} \). The derivation \( P \vdash_{st} \neg r \dashv P \vdash_{st} \true \) does exist because, considering the clause defining \( r \) in \( P \) and condition (a) in Definition 4.2.2, we have that \( P \vdash_{st} \neg (Q_1 \supset \neg p(x)) \Rightarrow_{st} x = a, P \vdash_{st} \neg (Q_2 \supset \neg q(x)) \Rightarrow_{st} x \neq a \) and \( \FET_{\Sigma} \models (\true \rightarrow x = a \lor x \neq a)^{\overline{x}} \).

The following example shows a failed derivation.

**Example 4.4.** Let \( P = \{ (p(x) \leftarrow \overline{x} = a) \} \) and \( Q = \{ (p(x) \leftarrow \neg r(x)); (r(x) \leftarrow \overline{x} = a) \} \). Starting a derivation from the sequent \( P \vdash_{st} Q \vdash_{st} \neg p(x) \) we cannot obtain a computed answer. The reason is that \( P|Q \vdash_{st} r(x) \dashv P|Q \vdash_{st} \overline{x} = a \), therefore \( P|Q \vdash_{st} \neg p(x) \dashv P|Q \vdash_{st} \overline{x} = a \) and \( P \vdash_{st} \neg p(x) \dashv P \vdash_{st} \overline{x} \neq a \). Hence, condition (b) in Definition 4.2.2 does not hold because there is not a satisfiable constraint less general than \( x = a \land x \neq a \).

### 4.2. A transformational semantics

In this section we define the semantics of extended programs in terms of a translation into the class of (standard) normal programs. This approach has several advantages. On the one hand, we can (indirectly) provide a declarative semantics of extended programs, without having to use a more complex logic (see, e.g. [16] where a modal logic is used). In particular, it is enough to consider the declarative semantics of the translated program. On the other hand, this transformational semantics is easy to implement. This means that we can easily build this kind of extension on top of a standard logic programming language.

The idea underlying this translation is quite simple. We rename all the predicates inside the implications to new fresh names. In addition, we add rules of the form \( p_i(x) \leftarrow p_j(x) \) where \( p_i \) and \( p_j \) are the names for the same predicate \( p \) in the program units \( P_i \) and \( P_j \), respectively, and where \( P_j \) includes \( P_i \).

Hereafter we assume that programs are defined over the signature \( \Sigma = \langle FS, PS \rangle \). In the following definitions \( PS' \) is a denumerable set of “fresh” predicates, that is, \( PS \cap PS' = \emptyset \). Moreover, we denote by \( C_{\Sigma} \) (resp. \( G_{\Sigma} \)) the set of all normal \( \Sigma \)-clauses (resp. \( \Sigma \)-goals), where embedded implications may occur. Similarly, we denote by \( C_{\Sigma} \) (resp. \( G_{\Sigma} \)) the set of all normal \( \Sigma \)-clauses (resp. \( \Sigma \)-goals), where no embedded implications occur.
Definition 4.4. Let $P$ be a program. Then, a renaming for predicates with respect to $P$ is a substitution of the form $\sigma : PS \cup PS' \rightarrow PS \cup PS'$, such that for all $p \in PS \cup PS'$

\[
\sigma(p) = \begin{cases} 
\text{gen}(p) & \text{if } \text{Def}(P, p) \neq \emptyset \\
 p & \text{otherwise}
\end{cases}
\]

(1)

where gen : $PS \cup PS' \rightarrow PS'$ is a function such that whenever it is applied returns a new predicate symbol in $PS'$ never used before. This definition is extended in order to apply a renaming to goals, clauses and programs as follows $\sigma(p(x)) = \sigma(p)(x), \sigma(\neg(p(x))) = \neg(\sigma(p)(x)), \sigma(P \triangleright G) = \sigma(P) \triangleright \sigma(G), \sigma((G_1, \ldots, G_m)) = (\sigma(G_1), \ldots, \sigma(G_m)), \sigma(p(x) \leftarrow \overline{G} \triangleright c) = \sigma(p)(x) \leftarrow \sigma(\overline{G}) \triangleright c,$ and $\sigma((C_1, \ldots, C_n)) = (\sigma(C_1), \ldots, \sigma(C_n))$.

Notice that the function gen is a generator of “fresh” predicate symbols. Also, we can compose $\sigma_1 \sigma_2 \ldots \sigma_k(p)$ even if each $\sigma_i$, for $i \in \{1, \ldots, k\}$, is not necessarily defined with respect to the same program. Moreover, the definition of gen ensures that a never used before predicate symbol will be obtained each time.

For the sake of simplicity we adopt the variable substitution notation. This means we denote $\sigma(p)$ as $p \sigma$ and similarly for goals, clauses and programs. Also, we denote the composition of renamings $\sigma_1 \ldots \sigma_i$ as $\overrightarrow{\sigma_i}$.

Definition 4.5. The translation function $T : \mathcal{P}(C_{\Sigma'}^0) \rightarrow \mathcal{P}(C_{\Sigma'}')$, where $\Sigma' = \langle FS, PS \cup PS' \rangle$ is defined in terms of the functions $T_\kappa : C_{\Sigma'}^0 \rightarrow C_{\Sigma'}'$ and $T_\gamma : G_{\Sigma'}^0 \rightarrow G_{\Sigma'}' \times \mathcal{P}(C_{\Sigma'}')$ as follows. For every program $P$,

\[
T(P) = \emptyset \text{ if } P = \emptyset \\
T(P) = \bigcup_{i=1}^n T_\kappa(C_i) \text{ if } P = \{C_1, \ldots, C_n\}
\]

such that

1. $T_\kappa(p(x) \leftarrow \overline{G} \triangleright c) = \{p(x) \leftarrow \overline{G'} \triangleright c\} \cup P'$ where $T_\gamma(G) = \langle \overline{G'}, P' \rangle$
2. If $G = \text{true}$ then $T_\gamma(G) = \langle \text{true}, \emptyset \rangle$
3. If $G = p(x)$ then $T_\gamma(G) = \langle p(x), \emptyset \rangle$
4. If $G = \neg(p(x))$ then $T_\gamma(G) = \langle \neg(p(x)), \emptyset \rangle$
5. If $G = Q \triangleright G_0$ then $T_\gamma(G) = \langle G', T(Q \sigma_G) \cup Q' \cup \text{ext}(\sigma_G) \rangle$ where
   (a) $\sigma_G$ is a renaming with respect to $Q$
   (b) $T_\gamma(G_0 \sigma_G) = \langle \overline{G'}, Q' \rangle$
   (c) $\text{ext}(\sigma_G) = \{p \sigma_G(x) \leftarrow p(x) \mid \sigma_G(p) \neq p \wedge \text{Def}(Q, p) \neq \emptyset\}$
6. For all extended goals $G, G'$ occurring in $P$, if $G \neq G'$ then for each $p$ in $PS$ either $\sigma_G(p) \neq \sigma_G'(p)$ or $\sigma_G(p) = \sigma_G'(p) = p$

This definition is extended to $\overline{G} = (G_1, \ldots, G_m)$ as follows

$T_\gamma(\overline{G}) = \langle (G_1', \ldots, G_m'), \bigcup_{i=1}^m P_i' \rangle$ where for each $i$ in $\{1, \ldots, m\}$ $T_\gamma(G_i) = \langle G_i', P_i' \rangle$
Notice that the set \( \text{ext}(\sigma) \) links renamed predicates with their “old” names in such a way that innermost-outermost visibility is preserved. The following example illustrates how the translation algorithm works.

**Example 4.5.** Let \( P = \{(p(x) \leftarrow \Box x = a); (q \leftarrow \{(p(x) \leftarrow \Box x = b) \supset \neg p(x))\}\} \). The translation of the program \( P \) is the following:

1. To translate a program means translating each of its clauses by \( T_\kappa \). That is, \( T(P) = T_\kappa(p(x) \leftarrow \Box x = a) \cup T_\kappa(q \leftarrow \{(p(x) \leftarrow \Box x = b) \supset \neg p(x))\} \).

2. The translation of the first clause is direct since its goal \( G \) is true (see cases 1. and 2. in Definition 4.5). \( T_\kappa(p(x) \leftarrow \Box x = a) = \{p(x) \leftarrow \Box x = a\} \).

3. The second clause is of the form \( q \leftarrow G \sqsupset c \) for \( c = \text{true} \) and \( G \) being of the form \( Q \supset G_0 \) for \( Q = \{p(x) \leftarrow \Box x = b\} \) and \( G_0 = \neg p(x) \). Then its translation is \( T_\kappa(q \leftarrow \{(p(x) \leftarrow \Box x = b) \supset \neg p(x))\} = \{q \leftarrow G' \} \cup P' \).

4. Finally, the translation of \( P \) is:
\[
T(P) = \{p(x) \leftarrow \Box x = a; q \leftarrow \neg p_1(x); p_1(x) \leftarrow \Box x = b; p_1(x) \leftarrow p(x)\}
\]

Since derivation steps are defined in terms of sequents, including stacks of programs, we need to extend the function \( T \).

**Definition 4.6.** Let \( S \) be the set of all possible sequences of programs and let \( S_k = P_1 \ldots | P_k, k \geq 1 \) be a specific sequence. Then, \( \hat{T} : S \rightarrow \mathcal{P}(\mathcal{C}_\Sigma) \) is inductively defined, as follows:

1. \( \hat{T}(P_1) = T(P_1) \)

2. \( \hat{T}(S_{k-1}|P_k) = \hat{T}(S_{k-1}) \cup T(P_k \sigma_k) \cup \text{ext}(\sigma_k) \) where \( \sigma_1 \) is the identity renaming and \( \sigma_k, k > 1 \), is a renaming with respect to \( P_k \sigma_{k-1} \).

The following example illustrates how \( \hat{T} \) works.

**Example 4.6.** Let \( P_1 = \{(p \leftarrow q)\} \) and \( P_2 = \{(s); (p \leftarrow \neg t); (p \leftarrow s)\} \)
\[
\hat{T}(P_1|P_2) = \hat{T}(P_1) \cup T(P_2 \sigma_1 \sigma_2) \cup \text{ext}(\sigma_2) \) where \( \sigma_2 \) is a renaming with respect to \( P_2 \sigma_1 \). From Definition 4.6 is easy to see that
\[
\hat{T}(P_1|P_2) = \{(p \leftarrow q)\} \cup (s \sigma_2); (p \sigma_2 \leftarrow \neg t); (p \sigma_2 \leftarrow s \sigma_2)\} \cup \{(s \sigma_2 \leftarrow s); (p \sigma_2 \leftarrow p)\}
\]
4.3. Soundness and completeness

In this section we prove the soundness and completeness of our transformational semantics. We do this by showing that the operational semantics defined in Section 4.1 is equivalent to the SLDFA operational semantics of the transformed programs and goals. Actually, we do this proof not using the original SLDFA semantics, as defined in [9], but a slightly simpler, but equivalent version. More precisely, this version of the SLDFA semantics just coincides with the semantics presented in Definition 3.1 when applied to programs and goals not including embedded implications. Anyhow, since Drabent [9] already proved that SLDFA-resolution of a program $P$, is sound and complete with respect to the 3-valued completion of the given program [19], and Fages [10] and laterLucio, Orejas and Pino [22] showed the equivalence of a least fixpoint semantics with the logical semantics defined by this completion, we may consider that, altogether, we implicitly have a full sound and completeness proof.

**Definition 4.7.** The derivation relation $\Rightarrow$ over sequents is defined as follows:

1. $P \vdash_{SLDFA} \overline{G}, p(\xi) \Rightarrow_{G2\otimes} P \vdash_{SLDFA} \overline{G}, G_2 \otimes c'$ if there exists a (renamed apart) clause $p(\xi) \leftarrow G_2 \otimes d \in \text{Def}(P, p)$ such that $P \vdash_{SLDFA} G_2 \otimes d \Rightarrow_{P} P \vdash_{SLDFA} c'$.

2. $P \vdash_{SLDFA} G, p(\xi) \Rightarrow_{G2\otimes} P \vdash_{SLDFA} G, G_2 \otimes c'$ if for every (renamed apart) clause $p(\xi) \leftarrow G_1, \ldots, G_m \otimes d \in \text{Def}(P, p)$ there exists $J \subseteq \{1, \ldots, m\}$, not necessarily unique, such that for all $j \in J$, we have $P \vdash_{SLDFA} G_j \otimes d \Rightarrow_{P} P \vdash_{SLDFA} c_j$, and $\text{FET}_\Sigma \models (c' \rightarrow \neg d \vee \bigvee_{j \in J} d_j)^\gamma$.

Moreover, we say that $P \vdash_{SLDFA} \overline{G} \otimes c$ can be proved with computed answer $c'$, denoted $S \vdash_{SLDFA} \overline{G} \otimes c \Rightarrow_{fa} c'$, if $P \vdash_{SLDFA} G \otimes c \Rightarrow_{G2\otimes} S \vdash_{SLDFA} c'$, $\text{FET}_\Sigma \models c'^\gamma$, and $\text{FET}_\Sigma \models (c' \rightarrow c)^\gamma$.

Finally, we can show the soundness and completeness of the transformation:

**Theorem 4.1** (Soundness). Let $S_k = P_1 | \ldots | P_k, k \geq 1$ and $\overline{G}$ be a stack of programs and a goal, respectively. If $T_\gamma(\overline{G} \sigma_k) = (\overline{G}', P')$ and $\hat{T}(S_k) \cup P' \vdash_{SLDFA} \overline{G} \otimes c' \Rightarrow_{fa} c'$, then $S_k \vdash_{st} \overline{G} \otimes c \Rightarrow_{st} c$.

**Theorem 4.2** (Completeness). Let $S_k = P_1 | \ldots | P_k, k \geq 1$ and $\overline{G}$ be a stack of programs and a goal, respectively. If $S_k \vdash_{st} \overline{G} \otimes c \Rightarrow_{st} c'$ and $T_\gamma(\overline{G} \sigma_k) = (\overline{G}', P')$, we have $\hat{T}(S_k) \cup P' \vdash_{SLDFA} \overline{G} \otimes c \Rightarrow_{fa} c'$.

5. Related work

As said in the introduction, the idea to use embedded implications for structuring logic programs is due to Miller [26]. In particular, Miller considered a class of programs close to Harrop formulas [27] where, in addition to embedded implications in goals, disjunctive goals and explicitly quantified goals are allowed. However, the use of implications in goals had already been proposed by Gabbay and Reyle [13] in the framework of an extension of Prolog, called
N-Prolog, as an approach to deal with hypothetical reasoning. Then, Giordano, Martelli and Rossi [17] established the distinction between the two possible scope rules. In particular, the algebraic semantics proposed in [17] is basically the one proposed by Miller but extended to interpret the clausal implication in a classical way. Considering classical and intuitionistic implications together rises some problems because the new semantics fits neither in classical logic nor in intuitionistic logic. This has been addressed in [2, 21] where a complete logic (extending classical first order logic with intuitionistic implication) is presented as the underlying logic of such static programming language.

Concerning the combination of negation and embedded implication, all the known approaches are related to what we consider dynamic scoping semantics. In the first known work, Gabbay [12] studied the logical properties of the language N-Prolog. He proposed intuitionistic logic as a suitable underlying logic and he used Kripke-like structures for giving model-based semantics. Moreover, soundness and completeness of N-Prolog with respect to intuitionistic logic was proved. Gabbay also introduced negation as failure into N-Prolog and pointed out that, since the success of a goal with respect to a program means intuitionistic provability, then failure means intuitionistic unprovability. The operational semantics for this new class of programs was a direct extension of negation as finite failure with rules for dealing with embedded implications occurring in goals (the deduction theorem). As one could expect, at this point problems arose because negation as finite failure does not coincide with intuitionistic negation. Basically, the difficulties observed by Gabbay concern the non-monotonic nature of the negation. In particular, in his proposal modus ponens did not hold and implication is non-transitive. Indeed, the problem with adding negation is the increasing of the set of clauses because a finite failure will depend on the considered set. Thus, a problem that Gabbay left open was to find a semantics where some form of modus ponens and transitivity of implication hold.

Following another line of research, McCarty [25] proposed a language for legal discourse based on what he called clausal intuitionistic logic [23]. The class of programs considered by him admitted the coexistence of embedded implication and negation in the right-hand side of clauses. Moreover, clauses were interpreted in an intuitionistic way. More precisely, McCarty addressed the model-based and the fixpoint semantics. The models were Kripke-like structures and he defined a monotonic immediate consequence operator but the continuity of this operator could not be concluded. Therefore, the constructibility could not be proved. In another paper [24], McCarty presented a proof procedure for this logic based on tableaux. The soundness and completeness of the proof procedure with respect to the fixpoint semantics was proved.

Bonner and McCarty [4] also extended logic programs with embedded implication and negation as failure. They only considered the restricted class of function-free stratifiable programs. The model theory developed was based on perfect model semantics [31] considering non monotonic Kripke-like structures. Soundness and completeness of this semantics were proved. More recently, Giordano and Olivetti [18] dealt with the dynamic class of programs. In the notion of failure that they used still remained the possibility of floundering. As a con-
sequence they had to impose some syntactic restrictions on programs in order to obtain completeness. They used Kripke-like models where the interpretations associated to a given world is a three-valued structure and the order relation between worlds is not monotonic. We can see that in order to achieve completeness results in the latter two approaches monotonicity of Kripke structures was relaxed and syntactic restrictions were imposed to the language.

Following a different approach, Freitag [11] proposed a transformational semantics for the class of logic programs with negation as failure and embedded implication. He defined a preprocessing mechanism for translating stratifiable function free logic programs into flat ones. The model-theoretic semantics was given in terms of perfect models based on non-monotonic Kripke-like structures. Additionally, soundness and completeness of the translated programs with respect to the model semantics considered were proved. The interpretation given to the embedded implication induced the dynamic scoping rule. However, the translation to flat programs was possible since stratifiable programs allows us to emulate the context characterization considered in those languages with static visibility rule. As we can see in previous work, all the attempts for extending normal logic programming with embedded implication considered negation as finite failure. The rule for combining dynamic logic programs with finite failure is the following

\[ P \cup Q \vdash_{dy} \neg p \]
\[ P \vdash_{dy} Q \supset \neg p \]

and the sequent \( P \cup Q \vdash_{dy} \neg p \) is considered an initial sequent when for each clause \( p \leftarrow G \) there exists a finitely failed tree of \( G \) with respect to \( P \cup Q \).

6. Conclusion

In this paper, we have provided semantics for the class of dynamic normal logic programs as well as for the class of static normal logic programs. In particular, we have considered structured normal languages in which the negation mechanism is constructive negation. More precisely, for the dynamic language we have followed the classical approach. This means we have first proposed an operational semantics. Then, we have defined a model-theoretic semantics in terms of a sort of ordered structures. In particular, these ordered structures are an adaptation of the intuitionistic Beth structures. Finally, an (effective) fix-point semantics has been provided and we have proven the equivalence between the semantics mentioned above. In order to deal with the static language, we have first defined an operational semantics and then we have presented an alternative semantics in terms of a transformation of the given structured program into a flat one. We have finished by showing that this transformation preserves the computed answers of the given static program.

We must note that, even if in this paper we provide a semantics to constructive negation in terms of intuitionistic models, this negation does not coincide with standard intuitionistic negation. In this sense, we consider that it may be
interesting to study the relation between the two negation from a logical point of view.

Acknowledgements. We would like to thank the reviewers for their detailed reports and their suggestions that have contributed to improve considerably this paper.

References


A Appendix

Proof of Theorem 3.1. The reflexive, antisymmetric and transitive properties are straightforward. Now, to prove that $\preceq_F$ is complete, consider that the bottom is $\bot = \langle W_\bot, \preceq_\bot, I_\bot \rangle$, where $W_\bot = \{(P, \emptyset) \mid P \text{ is a } \Sigma\text{-program}\}$, $\preceq_\bot = \{(w, w) \mid w \in W_\bot\}$ and $\forall w \in W_\bot, I_\bot (w) = \{\text{true}\}$.

The least upper bound for every increasing chain of linear PC structures $A_0 \preceq_F A_1 \preceq_F \ldots$, can be described as follows:

$$A_\downarrow = \langle W_\downarrow, \preceq_\downarrow, I_\downarrow \rangle$$

(2)

where

1. $W_\downarrow = \{(P, L_j^w) \mid P \text{ is a } \Sigma\text{-program, } j \in \mathbb{N}, L_j^w = \bigcup_{i \in \mathbb{N}} L_j^{A_i} \text{ and } (P, L_j^{A_i}) \in W_{A_i}\}$
2. $\forall j \in \mathbb{N}, \forall (P, L_j^w) \in W_\downarrow : I_\downarrow((P, L_j^w)) = \bigcup_{i \in \mathbb{N}} I_{A_i}((P, L_j^{A_i}))$

Finally, we have to prove that $A_\downarrow$ is the least upper bound. That is, if there exists a linear PC structure $A$ such that $\forall i \in \mathbb{N} : A_i \preceq_F A$, then $A_\downarrow \preceq_F A$. Let $A$ be a linear PC structure such that $\forall i \in \mathbb{N} : A_i \preceq_F A$. Hence, by Definition 3.13, for each $i \in \mathbb{N}$, for each $j \in \mathbb{N}$, for each $(P, L_j^{A_i}) \in W_{A_i}$, and for each $(P, L_j^A) \in W_A$ we have that $L_j^{A_i} \subseteq L_j^A$ and $I_{A_i}((P, L_j^{A_i})) \subseteq I_A((P, L_j^A))$. This implies that for each $j \in \mathbb{N}$: $\bigcup_{i \in \mathbb{N}} L_j^{A_i} \subseteq L_j^A$ and $\bigcup_{i \in \mathbb{N}} I_{A_i}((P, L_j^{A_i})) \subseteq I_A((P, L_j^A))$. Therefore, $L_j^\downarrow \subseteq L_j^A$ and $I_\downarrow((P, L_j^\downarrow)) \subseteq I_A((P, L_j^A))$. Consequently, $A_\downarrow \preceq_F A$.

To prove Theorem 3.2, we first prove the following Lemma:

Lemma 6.1. For any $\Sigma$-program $P$ and any linear PC structure $A$, $\Pos_P^A$ is a monotonic and continuous operator with respect to $\preceq_F$.

Proof. To prove monotonicity we have to prove the following. Let $P$ be a $\Sigma$-program and let $A$ and $B$ two linear PC structures such that $A \preceq_F B$. Then, for each $\Sigma$-program $P'$, considering its associated (possibly infinite) chains $(P', L_0^A) \preceq \ldots \preceq (P', L_i^A) \preceq \ldots$ in $A$ and $(P', L_0^B) \preceq \ldots \preceq (P', L_j^B) \preceq \ldots$ in $B$, $i \in \mathbb{N}$, we have that $(P', L_i^A) \preceq (P', L_j^B)$ and $I_A((P', L_i^A)) \subseteq I_B((P', L_j^B))$. Therefore, by definition of $\Pos_P^A$ and $\Pos_P^B$, and by the monotonicity of logical consequence, $\Pos_P^A(w) \subseteq \Pos_P^B(w)$.

Since $\Pos_P^A$ is monotone, to prove continuity it is enough to prove that for any infinite chain of $\Sigma$-structures $A_0 \preceq_F A_1 \preceq_F \ldots$, we have that, for each $\Sigma$-program $P'$, for each $w \in W_{\bigcup_{i \in \mathbb{N}} A_i}$: $\Pos_{P^{i+1}}^{\bigcup_{i \in \mathbb{N}} A_i}(w) \subseteq \bigcup_{i \in \mathbb{N}} \Pos_{P^i}^{A_i}(w)$. Let $w = (P', L_j) \in W_{\bigcup_{i \in \mathbb{N}} A_i}$, $j \in \mathbb{N}$ and let $p \forall c \in \Pos_{P^{i+1}}^{\bigcup_{i \in \mathbb{N}} A_i}(w)$. Therefore, there is $\{p(x) \leftarrow G^k b^k \mid 1 \leq k \leq n\} \subseteq P' \cup P'$, and for each $k$, $1 \leq k \leq n$, $(P', L_j), \forall_i A_i \models_{L_i} G^k b^k$ and $FET_{\Sigma \cup P'} = (c \rightarrow \bigvee_{k=1}^n d^k)^x$. Since $W_{\bigcup_{i \in \mathbb{N}} A_i} = \langle W_{\bigcup_{i \in \mathbb{N}} A_i}, \preceq_{\bigcup_{i \in \mathbb{N}} A_i}, I_{\bigcup_{i \in \mathbb{N}} A_i} \rangle$...
First of all, monotonicity is proved by showing that for all linear PC structure \( A \) and \( B \) such that \( A \preceq_F B \) then \( T_P(A) \preceq_F T_P(B) \). This is, we have to prove that, for each \( \Sigma \)-program \( P' \), considering its associated chains \( (P', L_{0}^{T_P(A)}) \leq (P', L_{1}^{T_P(A)}) \leq \ldots \leq (P', L_{i}^{T_P(A)}) \leq \ldots \) in \( T_P(A) \), and \( (P', L_{0}^{T_P(B)}) \leq (P', L_{1}^{T_P(B)}) \leq \ldots \leq (P', L_{i}^{T_P(B)}) \leq \ldots \) in \( T_P(B) \), \( i \in \mathbb{N} \), the following two properties hold:

1. \( \forall i \in \mathbb{N} : L_{i}^{T_P(A)} \subseteq L_{i}^{T_P(B)} \). By definitions \( Neg_P^A \) and \( Neg_P^B \), and since \( I_{T_P(A)}((P',L_{i}^{T_P(A)})) \subseteq I_{T_P(B)}((P',L_{i}^{T_P(B)})) \). By Definition 3.12:
   - (a) For each \( \Sigma \)-program \( P' \), \( I_{T_P(A)}((P',\emptyset)) = Pos_P^A((P',\emptyset)) \), and \( \forall w \neq (P',\emptyset) \in W_T(A) : I_{T_P(A)}(w) = \bigcup_{v \in W_A \land w = Neg_P^A(v)} Pos_P^A(v) \).
   - (b) For each \( \Sigma \)-program \( P' \), \( I_{T_P(B)}((P',\emptyset)) = Pos_P^B((P',\emptyset)) \), and \( \forall w \neq (P',\emptyset) \in W_T(B) : I_{T_P(B)}(w) = \bigcup_{v \in W_B \land w = Neg_P^B(v)} Pos_P^B(v) \).

Since \( A \preceq_F B \), by definitions \( Pos_P^A \) and \( Pos_P^B \), and by Lemma 6.1, we have that for each \( \Sigma \)-program \( P' \), \( \forall i \in \mathbb{N} : I_A((P',L_{i}^A)) \subseteq I_B((P',L_{i}^B)) \).

Consequently, for each \( \Sigma \)-program \( P' \), \( I_{T_P(B)}((P',\emptyset)) = Pos_P^B((P',\emptyset)) \).

Hence, \( T_P(A) \preceq_F T_P(B) \).

Now, to prove continuity, let us consider that for any infinite chain of linear PC structures \( A_i \preceq_F A_i \preceq_F \ldots \) its least upper bound \( \bigcup_{i \in \mathbb{N}} A_i = A_{\cup} \), where \( A_{\cup} \) is defined in equation (2) , in the proof of Theorem 3.1.

Then, since \( T_P \) is monotonic, we have that \( \bigcup_{i \in \mathbb{N}} T_P(A_i) \preceq_F T_P(\bigcup_{i \in \mathbb{N}} A_i) \), so, to prove continuity it is enough to prove \( T_P(\bigcup_{i \in \mathbb{N}} A_i) \preceq_F \bigcup_{i \in \mathbb{N}} T_P(A_i) \). This is, \( T_P(A_{\cup}) \preceq_F \bigcup_{i \in \mathbb{N}} T_P(A_i) \).

Now, since \( T_P \) is monotonic we can consider the infinite chain of linear PC structures \( T_P(A_0) \preceq_F T_P(A_1) \preceq_F \ldots \), where its least upper bound, \( T_P(\bigcup_{i \in \mathbb{N}} A_i) \) is defined in analogous way to \( A_{\cup} \), as follows:

\[
T_P(\bigcup_{i \in \mathbb{N}} A_i) = (W_{T_P(\bigcup_{i \in \mathbb{N}} A_i)}, I_{T_P(\bigcup_{i \in \mathbb{N}} A_i)})
\]

where

1. \( W_{T_P(\bigcup_{i \in \mathbb{N}} A_i)} = \{(P', L_{j}^{T_P(\bigcup_{i \in \mathbb{N}} A_i)}) \mid P' \in \Sigma \text{-program}, j \in \mathbb{N}, L_{j}^{T_P(\bigcup_{i \in \mathbb{N}} A_i)} = \bigcup_{i \in \mathbb{N}} L_{j}^{T_P(A_i)} \}

2. \( \forall j \in \mathbb{N}, \forall (P', L_{j}^{T_P(\bigcup_{i \in \mathbb{N}} A_i)}) \in W_{T_P(\bigcup_{i \in \mathbb{N}} A_i)} : I_{T_P(\bigcup_{i \in \mathbb{N}} A_i)}((P', L_{j}^{T_P(\bigcup_{i \in \mathbb{N}} A_i)})) = \bigcup_{i \in \mathbb{N}} I_{T_P(A_i)}((P', L_{j}^{T_P(A_i)})) \)
Actually, we have to prove that, for each $\Sigma$-program $P'$, considering its associated chain $(P', L_{0}^{TP(A_{\Sigma})}) \leq (P', L_{1}^{TP(A_{\Sigma})}) \leq \ldots \leq (P', L_{j}^{TP(A_{\Sigma})}) \leq \ldots$ in $TP(A_{\Sigma})$, and $(P', L_{0}^{TP}) \leq (P', L_{1}^{TP}) \leq \ldots \leq (P', L_{j}^{TP}) \leq \ldots$ in $TP \cup \forall \in \mathbb{N}$, the following two properties hold:

1. $L_{j}^{TP(A_{\Sigma})} \subseteq L_{j}^{TP}$. By definition of $A_{\Sigma}$, there exists $i \in \mathbb{N}$ such that $(P', L_{i}^{TP(A_{\Sigma})}) = (P', L_{i}^{TP(A_{\Sigma})})$. This is, $L_{j}^{TP(A_{\Sigma})} = L_{j}^{TP(A_{\Sigma})}$. Therefore, by definition of $TP_{\cup}$ in (3), $L_{j}^{TP(A_{\Sigma})} \subseteq L_{j}^{TP}$. Hence, $L_{j}^{TP(A_{\Sigma})} \subseteq L_{j}^{TP}$.

2. $IT_{TP}(A_{\Sigma})(I'(P', L_{i}^{TP(A_{\Sigma})})) \subseteq IT_{TP_{\cup}}(I'(P', L_{i}^{TP})).$ By definition of $A_{\Sigma}$, there exists $i \in \mathbb{N}$ such that $IT_{TP}(A_{\Sigma})(I'(P', L_{i}^{TP(A_{\Sigma})})) = IT_{TP}(A_{\Sigma})(I'(P', L_{i}^{TP(A_{\Sigma})}))$. Thus, $IT_{TP}(A_{\Sigma})(I'(P', L_{i}^{TP(A_{\Sigma})})) \subseteq \bigcup_{i \in \mathbb{N}} IT_{TP}(A_{\Sigma})(I'(P', L_{i}^{TP(A_{\Sigma})}))$. Hence, by definition of $TP_{\cup}$ in (3) we have that $IT_{TP}(A_{\Sigma})(I'(P', L_{i}^{TP(A_{\Sigma})})) \subseteq IT_{TP_{\cup}}(I'(P', L_{i}^{TP})).$

Finally, as a consequence of the Knaster-Tarski theorem [36], $TP$ has a least fix-point $TP_{\omega}$.

Proof of Proposition 3.1. Let $P$ be $\Sigma$-program. To prove that $TP_{\omega} \in Mod(P)$ we have to see that $TP_{\omega} \models P$ and this is $TP_{\omega}$ is a supported model of $P$. Thus, we have to prove that the following two conditions hold:

1. For each $w \in W_{TP_{\omega}}$, for each clause $p(\bar{x}) \leftarrow \bar{C}d \in P \cup P_{w}$, there exists a $\Sigma$-constraint $c$, $FET_{\Sigma} \models (c \rightarrow d)^{\uparrow}$, such that $w, TP_{\omega} \models p(\bar{x}) \leftarrow \bar{C}d$. Since $TP_{\omega}$ is a linear PC structure, it is enough to prove that for each $\Sigma$-program $P'$, for each clause $p(\bar{x}) \leftarrow \bar{C}d \in P \cup P'$, there exists a $\Sigma$-constraint $c$, $FET_{\Sigma} \models (c \rightarrow d)^{\uparrow}$, such that $(P', \emptyset), TP_{\omega} \models p(\bar{x}) \leftarrow \bar{C}d$. Actually, what we have to prove is that for each $\Sigma$-program $P'$, for each clause $p(\bar{x}) \leftarrow \bar{C}d \in P \cup P'$, if there exists $k \in \mathbb{N}$ such that there exists a $\Sigma$-constraint $d'$, $FET_{\Sigma} \models (d' \rightarrow d)^{\uparrow}$ and $(P', \emptyset), TP_{\omega} \models p(\bar{x}) \leftarrow \bar{C}d'$. Then there exists a $\Sigma$-constraint $c$, $FET_{\Sigma} \models (c \rightarrow d')^{\uparrow}$ and $(P', \emptyset), TP_{\omega} \models p(\bar{x}) \leftarrow \bar{C}d'$. Let $k \in \mathbb{N}$ such that there exists a $\Sigma$-constraint $d''$, $FET_{\Sigma} \models (d'' \rightarrow d)^{\uparrow}$ and $(P', \emptyset), TP_{\omega} \models p(\bar{x}) \leftarrow \bar{C}d''$. In particular, there exists a world $(P', L)$ such that $(P', L), TP_{\omega} \models p(\bar{x}) \leftarrow \bar{C}d''$. Therefore, by Definition 3.11, there exists a $\Sigma$-constraint $c$, $FET_{\Sigma} \models (c \rightarrow d)^{\uparrow}$, such that $p(\bar{x}) \leftarrow \bar{C}d$. Hence, by definition of $TP_{\omega}$, there exists a world $w, TP_{\omega} \models p(\bar{x}) \leftarrow \bar{C}d$. Additionally, by monotonicity of negative and positive information in worlds and, since $c$ is less general than $d'$ and for each world $w = (P_{w}, L_{w})$, $L_{w}$ and $I(w)$ are closed under less general constraint, $w, TP_{\omega} \models (P', \emptyset), TP_{\omega} \models p(\bar{x}) \leftarrow \bar{C}d$. Consequently, we have that $(P', \emptyset), TP_{\omega} \models p(\bar{x}) \leftarrow \bar{C}d$.

2. For each $w \in W_{TP_{\omega}}$, $w$ is a supported world. Straightforward from Definition 3.10
that one of the following two cases holds:

1. \( B(P_i) \subseteq T_P \uparrow \omega(P_i) \). Therefore, by Definition 3.14.1, \( W_B(P_i) = W_{T_P} \uparrow \omega(P_i) \) and \( \forall w \in W_B(P_i), I_B(P_i)(w) \subseteq I_{T_P} \uparrow \omega(P_i)(w) \). This means that \( T_P \uparrow \omega \) has more positive information than necessary. Consequently, it contradicts the fact that \( T_P \uparrow \omega \) is the least fix point of the immediate consequence operator \( T_P \).

2. There exist strict bars \( B \subseteq W_B \) and \( B' \subseteq W_{T_P} \uparrow \omega \) with respect to \( (P_i, \emptyset) \) in \( B(P_i) \) and \( T_P \uparrow \omega(P_i) \), respectively, such that \( B \uparrow \subseteq B' \downarrow \), and considering the \( \preceq \) increasing chains \( v_0, \ldots, v_k, w_1 \) and \( v_0, \ldots, v_k, w_2 \) in \( B(P_i) \) and \( T_P \uparrow \omega(P_i) \), respectively, where \( v_0 = (P_i, \emptyset) \) and, \( w_1 \in B \) and \( w_2 \in B' \), such that the following condition holds:

\[
I_B(w_1) \subseteq I_{T_P \uparrow \omega}(w_2) \text{ and } L_{w_2} \subseteq L_{w_1},
\]

Since \( B \uparrow \subseteq B' \downarrow \), this implies that, having both (sub) \( \Sigma \)-structures the same (negative and positive) information in \( v_k \), in a previous iteration of the \( T_P \) operator, either no new (supported) negative information was obtained or positive information was added to \( w_2 \) without being supported. This contradicts the fact that \( T_P \uparrow \omega \) is a supported model.

\[\square\]

Proof of Proposition 3.4. We proceed by reductio ad absurdum. Let us suppose there exists a \( \Sigma \)-structure \( B = (W_B, \preceq, I_B) \), such that \( B \) is the \( \preceq \)-least model in \( \text{Mod}(P) \). Therefore, \( B \sqsubseteq T_P \uparrow \omega \). Hence, by Definition 3.17, for each chain \( P_1 \subseteq \ldots \subseteq P_i \subseteq \ldots \) of \( \Sigma \)-programs, there exists \( i \in \mathbb{N} \) such that \( \forall j \in \mathbb{N}, j < i : B(P_j) = T_P \uparrow \omega(P_j) \) and \( B(P_i) \sqsubseteq T_P \uparrow \omega(P_i) \). Thus, by Definition 3.15, we have that one of the following two cases holds:

1. \( B(P_i) \subseteq T_P \uparrow \omega(P_i) \). Therefore, by Definition 3.14.1, \( W_B(P_i) = W_{T_P} \uparrow \omega(P_i) \) and \( \forall w \in W_B(P_i), I_B(P_i)(w) \subseteq I_{T_P} \uparrow \omega(P_i)(w) \). This means that \( T_P \uparrow \omega \) has more positive information than necessary. Consequently, it contradicts the fact that \( T_P \uparrow \omega \) is the least fix point of the immediate consequence operator \( T_P \).

2. There exist strict bars \( B \subseteq W_B \) and \( B' \subseteq W_{T_P} \uparrow \omega \) with respect to \( (P_i, \emptyset) \) in \( B(P_i) \) and \( T_P \uparrow \omega(P_i) \), respectively, such that \( B \uparrow \subseteq B' \downarrow \), and considering the \( \preceq \) increasing chains \( v_0, \ldots, v_k, w_1 \) and \( v_0, \ldots, v_k, w_2 \) in \( B(P_i) \) and \( T_P \uparrow \omega(P_i) \), respectively, where \( v_0 = (P_i, \emptyset) \) and, \( w_1 \in B \) and \( w_2 \in B' \), such that the following condition holds:

\[
I_B(w_1) \subseteq I_{T_P \uparrow \omega}(w_2) \text{ and } L_{w_2} \subseteq L_{w_1},
\]

Since \( B \uparrow \subseteq B' \downarrow \), this implies that, having both (sub) \( \Sigma \)-structures the same (negative and positive) information in \( v_k \), in a previous iteration of the \( T_P \) operator, either no new (supported) negative information was obtained or positive information was added to \( w_2 \) without being supported. This contradicts the fact that \( T_P \uparrow \omega \) is a supported model.

\[\square\]

Proof of Lemma 3.1. We proceed by induction on the number of derivation steps, \( k = n + \sum_{j=1}^{m} n_j \), where \( m \) is the number of subderivations, \( n \) the number of derivation steps in the main derivation, and \( n_j \) the number of derivation steps in the subderivation \( j \). The base step is when \( k = 0 \). So, what we have is that \( P \cup Q \vdash_{\text{dyn}} \varnothing \) and, by definition of \( \Sigma \)-structures, \( (Q, \emptyset), T_P \uparrow \omega \models \varnothing \).

Suppose that for all \( \Sigma \)-programs \( Q \), if \( P \cup Q \vdash_{\text{dyn}} \varnothing \models \varnothing \) with a number of derivation steps \( < k \), then \( (Q, \emptyset), T_P \uparrow \omega \models \varnothing \).

For simplicity, in the rest of the proof we assume, without loss of generality, that the derivation steps are induced by the first extended \( \Sigma \)-literals in the right part of sequents. Now we prove the statement for \( k = n + k' \). There are 4 cases to consider, depending on the derivation step:

1. \( P \cup Q \vdash_{\text{dyn}} p(\overline{x}), \overline{G} \models \varnothing \Rightarrow \varnothing \). Therefore, there exists a (renamed apart) clause \( p(\overline{x}) \leftarrow \overline{G} \models d \in \text{Def}(P \cup Q, p) \) such that \( FET_\Sigma \models (c \land d)^3 \) and \( P \cup Q \vdash_{\text{dyn}} \overline{G}, \overline{G} \models c \land d \vdash P \models \varnothing \). By the inductive hypothesis, \( (Q, \emptyset), T_P \uparrow \omega \models \overline{G}, \overline{G} \models c \) and, by Definition 3.6, \( (Q, \emptyset), T_P \uparrow \omega \models \overline{G} \models c \) and \( (Q, \emptyset), T_P \uparrow \omega \models \overline{G} \models c \). Thus, by Definition 3.6 and the definition of \( \text{Pos}_{T_P \uparrow \omega} \), there exists a world \( (Q, L) \in W_{T_P \uparrow \omega} \) such that \( (Q, L), T_P \uparrow \omega \models \models p(\overline{x}) \models c \). Hence, \( (Q, \emptyset), T_P \uparrow \omega \models p(\overline{x}) \models c \). Consequently, \( (Q, \emptyset), T_P \uparrow \omega \models p(\overline{x}), \overline{G} \models c \).

40
2. \( P \cup Q \vdash_{\text{dyn}} \neg p(\overline{x}), \overline{G} \vdash_{\text{syn}} c' \). Therefore, in any (renamed apart) clause \( p(\overline{x}) \leftarrow G_1, \ldots, G_m \circ d \in \text{Def}(P \cup Q, p) \) there exists \( J \subseteq \{1, \ldots, m\} \) such that \( \forall j \in J : P \cup Q \vdash_{\text{dyn}} \neg G_j \circ d \) with computed answers \( d_j \) and \( FET_* \vdash (c' \rightarrow \neg \bigvee_{j \in J} d_j)^y \) and, \( P \vdash_{\text{dyn}} \overline{G} \circ d' \vdash_{\text{syn}} P \vdash_{\text{dyn}} \circ c' \) (therefore, \( FET_* \vdash (c' \rightarrow d')^y \)). By inductive hypothesis, \( \forall j \in J : (Q, \emptyset), T_p \uparrow \omega \vdash \neg G_j \circ d_j \). Thus, by Definition 3.6 and the definition of \( \text{Neg}^{T_p}_{\text{syn}} \), there exists \( k \in \mathbb{N} \) such that there exists a world \( (Q, L) \in T_p[k] \) and \( p(\overline{x}) \circ c' \in L \). Hence, \( (Q, \emptyset), T_p \uparrow \omega \vdash \neg p(\overline{x}) \circ c' \). In addition, by induction we also have that \( (Q, \emptyset), T_p \uparrow \omega \vdash \neg p(\overline{x}), \overline{G} \circ c' \). Hence, \( (Q, \emptyset), T_p \uparrow \omega \vdash (Q' \supset G), \overline{G} \circ c' \).

Proof of Lemma 3.2. We will prove by induction on \( k \) that \( \forall k \geq 0 \) such that \( (Q, \emptyset), T_p \uparrow k \vdash \overline{G} \circ c \) implies \( P \cup Q \vdash_{\text{dyn}} \overline{G} \circ c \) can be proved with computed answers \( c_1, \ldots, c_n \) such that \( FET_* \vdash (c \rightarrow \bigwedge_{i=1}^n c_i)^y \).

The base step is \( k = 0 \). Then, for each satisfiable constraint \( c \), \( (Q, \emptyset), T_p \uparrow 0 \vdash \circ c \). Therefore, \( P \cup Q \vdash_{\text{dyn}} \circ c \Rightarrow_{\text{syn}} c \). Suppose that for every \( \Sigma \)-program \( Q \), if \( (Q, \emptyset), T_p \uparrow j \equiv \overline{G} \circ c \), \( j < k \) then \( P \cup Q \vdash_{\text{syn}} \overline{G} \circ c \equiv_{dyn} c_1, \ldots, c_n \) such that \( FET_* \vdash (c \rightarrow \bigwedge_{i=1}^n c_i)^y \). Now we prove it for \( k \). To do this, we have to consider the following cases:

1. Assume \( (Q, \emptyset), T_p \uparrow k \vdash p(\overline{x}) \circ c \). This implies that there exists a bar \( B \) with respect to \( (Q, \emptyset) \) such that for all \( v \in B : p(\overline{x}) \circ c \in I_{T_p[k]}(v) \). By definition of \( I_{T_p[1(k-1)]}(v) \), \( p(\overline{x}) \circ c \in P_{\text{syn}}^{T_p[1(k-1)]}(v) \). Therefore, there exists a world \( (Q, L) \in W_{T_p[1(k-1)]} \), there exists \( \{p(\overline{x}) \leftarrow \overline{G} \circ d' \ | 1 \leq i \leq m\} \subseteq \text{Def}(P \cup Q, p) \) and \( (Q, L'), T_p \uparrow (k-1) \vdash_{\text{syn}} \overline{G} \circ d' \vdash_{\text{syn}} P \vdash_{\text{syn}} \circ d' \). We have the following. On the one hand, by induction, \( P \cup Q \vdash_{\text{syn}} \overline{G} \circ d' \Rightarrow_{\text{syn}} d'_1, \ldots, d'_n \) such that \( FET_* \vdash (d' \rightarrow \bigwedge_{i=1}^n d'_i)^y \). On the other hand, since \( FET_* \vdash (c \rightarrow \bigwedge_{i=1}^n d'_i)^y \), for each definition of \( p \) we can construct the derivation \( P \cup Q \vdash_{\text{syn}} p(\overline{x}) \circ c \vdash_{\text{syn}} \overline{G} \circ c \wedge d' \vdash_{\text{syn}} \circ c_1 \). Consequently, \( P \cup Q \vdash_{\text{syn}} p(\overline{x}) \circ c \Rightarrow_{\text{syn}} c_1, \ldots, c_n \) such that \( FET_* \vdash (c \rightarrow \bigwedge_{i=1}^n c_i)^y \).

2. Assume \( (Q, \emptyset), T_p \uparrow k \vdash \neg p(\overline{x}) \circ c \). This implies that there exists a bar \( B \) with respect to \( (Q, \emptyset) \) such that for all \( v \in B : p(\overline{x}) \circ c \in L_v \). By definition of \( \text{Neg}^{T_p[1k]} \), for all (renamed apart) clause \( p(\overline{x}) \leftarrow G_1, \ldots, G_m \circ d \in \text{Def}(P \cup Q, p) \) there exists a world \( (Q, L) \in W_{T_p[1(k-1)]} \), \( L' \subseteq L_v \) and there exist satisfiable constraints \( \{d_j\}_{j \in J} \), \( J \subseteq \{1, \ldots, m\} \) such that \( \forall j \in J : \)}
We proceed by induction on the number of derivation steps, \( n \). Assume \( \{(Q, \emptyset), T_P \upharpoonright k \models Q' \supset G \} \). Without loss of generality we can consider the flattened extended \( \Sigma \)-expression of \( Q' \supset G \), say \( P_1 \cup \ldots \cup P_m \supset \emptyset \). This implies \( \{(Q \cup P_1 \cup \ldots \cup P_m, \emptyset), T_P \upharpoonright k \models \emptyset \} \). Since \( \ell \) is a normal literal this case can be reduced to previous two cases. Thus, it is easy to see that \( \{(Q \cup P_1 \cup \ldots \cup P_m, \emptyset), T_P \upharpoonright k \models \emptyset \} \).

4. Assume \( \{(Q, \emptyset), T_P \upharpoonright k \models G_i \cup \emptyset \} \). This implies \( \forall i \in \{1, \ldots, m\} : \{(Q, \emptyset), T_P \upharpoonright k \models G_i \cup \emptyset \} \). We proceed by case analysis on \( \mathcal{G} \).

1. \( \mathcal{G} = p(\pi) \). Then, there exists \( i \in \{1, \ldots, k\} \) and a (renamed apart) clause \( p(\pi) \leftarrow \mathcal{G} \supset d \in \text{Def}(P_i, p) \) such that \( S_i \vdash_{\text{st}} \mathcal{G} \supset d \Rightarrow_{\text{st}} c' \).

2. \( \mathcal{G} = \neg p(\pi) \). Then, for every \( i \in \{1, \ldots, k\} \) and for every (renamed apart) clause \( p(\pi) \leftarrow G_1, \ldots, G_m \supset d \in \text{Def}(P_i, p) \) there exists \( J \subseteq \{1, \ldots, m\} \) such that for each \( j \in J \), assuming \( G_j = Q_1 \cup \ldots \cup Q_n \cup \emptyset \) with \( 0 \leq \ell \), we have \( P_i \vdash_{\text{st}} \neg \ell \supset c' \Rightarrow_{\text{st}} d \), and \( \mathcal{F \text{ET}}_{\Sigma} = (c' \supset \neg d \lor \bigvee_{j \in J} d_j)^{\mathcal{G}} \).

Now, by induction, for every \( i \in \{1, \ldots, k\} \), for each clause \( p(\pi) \leftarrow G_1, \ldots, G_m \supset d \in \text{Def}(P_i, p) \) and for each \( j \in J \), if
Again, we proceed by induction on the number of derivation steps.

Proof of Theorem 4.2. We proceed by case analysis on $\mathcal{G}$.

3. $\mathcal{G} = Q \supset G_Q$. Then, $S_k \vdash_{st} Q \supset G_Q \mathcal{C}$ with computed answer $c'$ and the number of derivation steps used in this derivation is $n + 1$. Thus, $S_k | Q \vdash_{st} G_Q \mathcal{C} \Rightarrow c'$ in $n$ steps and, by induction, $\mathcal{T}(S_k(Q) \cup P^\prime_Q) \vdash_{SLDFA} G_Q' \mathcal{C}$ $\Rightarrow f_a c'$, where $\mathcal{T}_\gamma(Q \mathcal{C}) = (G_Q', P_Q')$. By definition of $\mathcal{T}$, $\mathcal{T}(S_k(Q)) = \mathcal{T}(S_k) \cup \mathcal{T}(Q \mathcal{C}) \cup ext(\mathcal{C})$, and, by definition of $\mathcal{T}_\gamma$, $\mathcal{T}_\gamma(Q \mathcal{C}) = (G_Q', P'^\prime)$, where $P'^\prime = \mathcal{T}(Q \mathcal{C}) \cup P_Q' \cup ext(\mathcal{C})$. Consequently, $\mathcal{T}(S_k(Q) \cup P'^\prime) \vdash_{SLDFA} G_Q' \mathcal{C} \Rightarrow f_a c'$, where $\mathcal{T}_\gamma(Q \mathcal{C}) = (G_Q', P'^\prime)$.

4. $\mathcal{G} = (G_1, \ldots, G_l)$, with $1 < l$. Therefore, for each $i \in \{1, \ldots, l\}$, $S_k \vdash_{st} G_i \mathcal{C} \Rightarrow c'_i$, $FET_{\mathcal{G}} = (c' \rightarrow c'_i)^m$ in a number of derivation steps smaller or equal to $n$. By induction, for every $i \in \{1, \ldots, l\}$, $\mathcal{T}(S_k) \cup P'_i \vdash_{SLDFA} G_i' \mathcal{C} \Rightarrow f_a c'_i$, where $T_\gamma\mathcal{C} = (G_i', P_i')$. Then, $\mathcal{T}(S_k) \cup \bigcup_{i=1}^l P'_i \vdash_{SLDFA} G_1' \ldots, G_l' \mathcal{C} \Rightarrow f_a c'_i$, since, by definition of $\mathcal{T}_\gamma$, $\mathcal{T}_\gamma(G_1, \ldots, G_l, \mathcal{C}) = (G_1', \ldots, G_l', \bigcup_{i=1}^l P'_i)$. Consequently, $\mathcal{T}(S_k) \cup \bigcup_{i=1}^l P'_i \vdash_{SLDFA} G_1' \ldots, G_l' \mathcal{C} \Rightarrow f_a c'_i$.

Proof of Theorem 4.2. Again, we proceed by induction on the number of derivation steps, $n$. The base step is when $n = 0$. In this case the theorem trivially holds.

Assume the theorem holds whenever the number of SLDFA-derivation steps is $\leq n$. We proceed by case analysis on $G$.

1. $\mathcal{G} = p(x)$. Then, $\mathcal{T}(\mathcal{C} \mathcal{P}_k) = (p \mathcal{P}_k(x), \emptyset)$ and there exists a (renamed apart) clause $p \mathcal{P}_k(x) \leftarrow \mathcal{C} d \in Def(\mathcal{T}(S_k), p \mathcal{P}_k)$ such that $e \cap d$ is a satisfiable constraint and $\mathcal{T}(S_k) \vdash_{SLDFA} \mathcal{C} \mathcal{P}_k \sim \mathcal{T}(S_k) \vdash_{SLDFA} \mathcal{C} d \Rightarrow f_a c'$. There are two cases:

(a) The clause $p \mathcal{P}_k(x) \leftarrow \mathcal{C} d$ is the transformation of a clause in $S_k$. That is, there exists a (renamed apart) clause $p(x) \leftarrow \mathcal{C}_d \in Def(P_1, p)$, for some $i \leq k$ and $\mathcal{T}_\gamma(\mathcal{C}_d) = (\mathcal{T}_\gamma(P_i'))$. Then, by induction, we have $S_i \vdash_{st} \mathcal{C}_d \Rightarrow f_a c'$ and, therefore, $S_k \vdash_{st} \mathcal{C} \mathcal{P}_k \sim \mathcal{T}(S_k) \vdash_{SLDFA} \mathcal{C} d \Rightarrow f_a c'$.

(b) The clause $p \mathcal{P}_k(x) \leftarrow \mathcal{C} d$ is an extension clause, i.e. a clause of the form $p \mathcal{P}_k(x) \leftarrow p \mathcal{P}_{k-1}(x)$, and $\mathcal{T}(S_k) \vdash_{SLDFA} p \mathcal{P}_{k-1}(x) \mathcal{C} \Rightarrow f_a c'$. But, by induction, this means $S_k \vdash_{st} p(x) \mathcal{C} \sim \mathcal{T}(S_k) \vdash_{SLDFA} \mathcal{C} d \Rightarrow f_a c'$.

2. $\mathcal{G} = \neg p(x)$. Let $p(x) \leftarrow G_1, \ldots, G_m \mathcal{C} d$ be a clause in $Def(P_1, p)$, for some $i$. Then, we have that its translation $\mathcal{P}_k(x) \leftarrow \ell_1, \ldots, \ell_m \mathcal{C} d$ is in $\mathcal{T}(S_k)$, where $\mathcal{T}_\gamma(G_1, \ldots, G_m, \mathcal{P}_k) = ((\ell_1, \ldots, \ell_m), \bigcup_{i=1}^m P_i), \bigcup_{i=1}^m P_i \subseteq \mathcal{T}(S_k)$. Hence, there exists $J \subseteq \{1, \ldots, m\}$ such that for each $j \in J$, $\mathcal{T}(S_k) \vdash_{SLDFA}$
\[-\ell_j \triangleright d \Rightarrow_{fa} d_j, \text{ and } FET_\Sigma \models (c' \rightarrow \neg d \lor \bigvee_{j \in J} d_j)^\forall. \] But, by induction, this means \( S_k \vdash_{st} \neg p(\overline{x}) \triangleright c \Rightarrow_{st} c'. \)

3. \( \overline{S} = Q_1 \supset \ldots \supset Q_m \supset \ell. \) Therefore, \( c' \) is an SLDFA-computed answer of \(-\ell \overline{\sigma}_{k+m} \triangleright c \) with respect to \( T_\gamma(S_k) \cup \bigcup_{i=1}^m Q'_i \) in \( n+1 \) derivations steps where \( T_\gamma(\overline{\sigma} k_m) = \langle \ell \overline{\sigma}_{k+m}, \bigcup_{i=1}^m Q'_i \rangle. \) Hence, there are two possibilities depending on the kind of literal \( \ell \overline{\sigma}_{k+m} \) (either positive or negative). Then, using similar arguments as in cases 1 and 2 of this proof, we obtain that \( S_k | Q_1| \ldots | Q_m \vdash_{st} \ell \overline{\sigma}_{k+m} \triangleright c \Rightarrow_{st} c'. \)

4. \( \overline{S} = (G_1, \ldots, G_l), \) with \( 1 < l. \) Therefore, for each \( i \in \{1, \ldots, l\}, \) \( T_\gamma(G_i, \overline{\sigma}_k) = \langle G'_i, P'_i \rangle, \) \( T_\gamma(S_k) \cup P'_i \vdash_{SLDFA} G'_i \triangleright c'_i, \) and \( FET_\Sigma \models (c' \rightarrow c'_i)^\forall \) in a number of derivation steps smaller or equal to \( n. \) By induction, for every \( i \in \{1, \ldots, l\} \) \( S_k \vdash_{st} G_i \triangleright c \Rightarrow_{st} c'_i, \) where \( T_\gamma(G_i, \overline{\sigma}_k) = \langle G'_i, P'_i \rangle. \) Then, \( S_k \vdash_{st} (G_1, \ldots, G_l) \triangleright c \Rightarrow_{st} c'. \)

\( \square \)