UNIVERSITÀ DEGLI STUDI DI PISA
DIPARTIMENTO DI INFORMATICA

DOTTORATO DI RICERCA IN INFORMATICA
Università di Pisa-Genova-Udine

Ph.D. Thesis: TD-3/94

Negation and Infinite Computations in Logic Programming

Alessandra Di Pierro

Abstract. This thesis is devoted to probe into some aspects of Logic Programming, namely the problem of inferring some negative information from a logic program, the denotational characterization of the meaning of a program, and the study of its infinite computations. Concerning the first problem, the attention is focused on the negative information expressed by the existential quantification of the negation of an atom (or a conjunction of atoms) which is *not* a logical consequence of the theory associated to a given logic program. Operationally, these formulas can be inferred by means of the *Negation As Instantiation* rule; from a logical point of view, they are shown to be exactly those formulas which are logical consequences of the theory obtained by considering the Clark completion of the given program on a universal vocabulary. The paradigm of Logic Programming is then extended by allowing the use of the existential negative expressions in the definition of the predicates, and, correspondingly, also the classical mechanism of SLD-resolution is extended so to deal with the new formulas.

The other two aspects are tackled within the more general framework of constraint logic programming. This language is reformulated in algebraic terms as the free language generated by a BNF grammar. This allows us to define a structured operational semantics, based on a transition system, by means of which a notion of observables is introduced, also including the results of infinite computations. Then, a compositional semantics based on logical operators is defined, and it is shown its full correspondence with the observables. The denotational meaning of the program is obtained by considering the domain of the upward-closed Scott-compact sets of constraints (the Smith powerdomain of the domain of constrains). It turns out that the well-known Negation As Failure of Logic Programming is just the Heyting negation on this domain.

March 1994

ADDR:Corso Italia 40,56125 Pisa,Italy. TEL:+39-50-887111 - TLX:590291 DIPISAI
E_MAIL: dipierro@di.unipi.it

Acknowledgments

My greatest debt is to Catuscia Palamidessi and Maurizio Martelli. As my advisors they have had a fundamental influence on the development of this work. Their advice greatly helped me also outside of the research field.

Part of this thesis would not exist if Catuscia had not aroused my interest in the subject of infinite computations. I also owe her all I know about constraints and concurrency and the crucial discussions solving any problem I encountered during my research. But, above all, I will be always grateful to Catuscia for her friendship and the encouragement that she provided me everytime I needed it to go on. I must also mention Giorgio Levi, one of the most generous persons I ever met.

Finally, I am very grateful to Wlodzimierz Drabent and Howard Blair for reviewing my thesis with rigor and for their enlightening comments which helped to improve the presentation. I alone am responsible for the imperfections and the mistakes that remain.

Contents

Chapter 1

Introduction

Although logic has been used as a tool for designing computers, and for reasoning about computers and computer programs since almost their beginning, the use of logic directly as a programming language, termed *logic programming*, dates back to 1974, when in his famous article [65], Kowalski proposed the use of predicate logic as a programming language. This proposal soon gained success, and logic programming become in a few years' time a very popular formalism. The reasons can be easily explained: the extreme simplicity, and (nevertheless) the formal rigor deriving from mathematical logic. For the latter logic programming constitutes the theoretical framework of PROLOG, the programming language developed by Colmerauer et al. in 1973 ([24]).

In the original formulation of Kowalski, logic programs are some particular formulas of first-order predicate logic, namely the so-called definite Horn formulas. From this point of view, they have two interpretations: as well-formed formulas or formulas deducible by the axioms and the rules of predicate logic system (corresponding to the notion of "formal validity" due to G. Frege), and as formulas which are true in all the interpretations of the predicate logic language (corresponding to the notion of "logical truth" due to Tarski). From a programming point of view, they have been provided with two kinds of semantics, operational and fixpoint, ([105]) which are the counterparts of the above mentioned interpretations in the framework of programming languages. In particular, the fixpoint semantics is a special case of (is equivalent to a) model-theoretic semantics.

The most important and pleasant feature of logic programming is the complete correspondence of the two kinds of semantics associated to a given program, namely the operational semantics and the declarative semantics (fixpoint and model-theoretic). In the light of the analogy with predicate logic, this correspondence becomes a special case of the Gödel completeness theorem, if (as suggested in [105]) we regard operational semantics as the syntactical derivability and declarative semantics as the logical validity of predicate logic.

Completeness is appealing also from a practical point of view, as it means that, in order to understand the result of a computation, no knowledge of the underlying computational mechanism (interpreter) is required. It is possible to establish the meaning of a computation simply by elementary mathematical tools. Of course, in making any extension of logic programming, it would be better to preserve this property.

Actually, the restriction to programs containing only definite Horn clauses has been felt soon to be too drastic for the logic programmers. Not long after the article of Van Emden and Kowalski a flurry of research started, aimed at extending logic programming in various directions. The realization that the introduction of aspects such as *negation*, *constraints* and *concurrency* indeed increases the expressive power of logic programming led to extensive studies in the three corresponding areas of research, which produced interesting results concerning both each single area and their interplays. However, the introduction of these new concepts has not been always painless, due to its effects more or less unpleasant with respect to the preservation of the declarative properties of logic programming.

The use of constraints in logic programming (clp) has been proposed by Jaffar and Lassez ([55, 54]) in order to allow a generic domain to be considered as the domain on which the computation is performed. This contrasts with the logic programming paradigm, where the domain is fixed (the Herbrand Universe), so that it is not possible to work directly in the intended domain of discourse; every semantic object has to be explicitly coded into a Herbrand term. Due to its flexibility, constraint logic programming does not represent a single language, but rather a scheme, CLP(X), which is parametric w.r.t. the particular domain of constraint, X, to embed within the logic programming shell. Any specific clp language is obtained by instantiating this schema. The benefits offered by this extension are without loss in semantic properties, as a satisfactory semantics similar to that for logic programs is achieved also for logic programs with constraints over some domain. It is important to notice that, from a theoretical point of view, the constraint logic programming paradigm provides a framework to unify several approaches for the integration of logic and functional programming ([50]).

Contemporary to the advent of constraint logic programming, another trend tried to enhance the expressivity of logic programming language by replacing the notion of sequential (and deterministic) execution with a concurrent one. The basic idea was to give a process-oriented interpretation of logic programs and goals as concurrent agents communicating through the logical variables seen as channels. Starting from the proposal of Clark and Gregory ([22]), an entire class of so-called concurrent logic languages based on the above idea, has been developed (see [91, 27] for a classification).

The two extensions of logic programming, constraints and concurrency, have been recently merged within a new paradigm: concurrent constraint programming (ccp, [88, 89]). This paradigm adopts an explicit syntax for expressing communication (*tell*), synchronization (*ask*), nondeterminism ($+$) and hiding ($\exists$). The model of computation in ccp is based on the idea that all processes share a common store which represents the constraint established up to that moment. Communication is achieved by adding (telling) some constraint to the store, and synchronization is achieved by checking (asking) if the store entails (implies) a given constraint.

The *ask* construct is the main difference between clp and ccp. It allows specification of choices which depend upon the environment. This makes problematic a simple extension of the semantics of clp. Thus, the advantages introduced by concurrency sacrifices the clean declarative nature of logic programming and constraint logic programming.

Another problematic extension is represented by *negation*. Negation appears as an aspect contrasting both the procedural and the declarative meaning of Horn clauses, for these can define only "positive" procedures and their models are sets containing only positive elements. In other words, a definite program can specify only positive information. Nevertheless, there are also some negative information which can be deducible from a positive logic program. Typically, in the context of deductive databases, positive information is explicitly (unit clauses) or implicitly (implicative clauses) represented in the database (positive logic program), whereas the negative one is coded as "everything that cannot be deduced".

Moreover, by allowing negative literals to occur in the body of the program clauses, the expressive power of logic programming grows considerably. As pointed out in [5], only by incorporating negation logic programming can become an appropriate formalism for expert system shells.

Unfortunately, as soon as *negation* comes on the scene, the original declarative nature of logic programming is not preserved, and the correspondence between the operational and the declarative meaning of a general program (i.e. a logic program whose clauses possibly contain negative literals in the body) is not so immediate any more. In particular, without suitable restrictions and hypothesis, it is impossible to obtain completeness.

Difficulties rise first of all from the intrinsically thorny nature of the concept of negation, which also comes out in daily life. When a human is asked whether a certain fact holds or not, and he answers '*no*', there are complex logical principles involved in arriving at the answer. The answer '*no*' does not mean the same in different contexts. There are several different negations, namely several ways of intending an answer of '*no*'. Examples of these can be found in [42] where, in particular, the *Negation as inconsistency* is analyzed: we can answer *no* to a question because if we say '*yes*' then we get undesirable results. But we could also answer '*no*' because we do not have positive information about the question (negation by default), or the question is specifically negated (strong negation) or we fail to retrieve it (negation as failure), and so on. The difficulties in the use of negation in computer systems are further complicated by computational considerations about complexity and efficiency, which could persuade us to prefer a certain kind of negation rather than another one. Moreover, some theoretical considerations about the completeness of some (correct) inference rule with respect to some theory could convince us that some logics is better than some other ones. In Chapter 6 we will give a survey of these problems and of the (many) solutions proposed in the literature since Clark introduced the matter by proposing a rule (the negation as failure,[23]) to infer negative facts from a logic program.

One of the main problem of negation as failure (and similar types of negation) has to do with *non-termination*. In fact, if the execution does not terminate, then it is not possible to decide the (absence of) success. This becomes a serious drawback when negation is incorporated into the programs (general programs), since it is not clear what should be the meaning of a program like $p \leftarrow \neg p$.

Solutions to this problem have been proposed in many directions, essentially consisting of:

- preventing the occurrence of non-termination by forbidding recursion in the definition of the program;
- allowing only those particular kinds of recursive definition which guarantee the possibility to define some "natural" model for the program;
- turning to the three-valued logics.

The first is not a quite satisfactory solution, since infinite behaviors are in many situations a desired feature of the program rather than a mistake or an obstacle to overcome. An adequate treatment of infinite computations is important mainly in connection with the above mentioned introduction in logic programming of the aspects related to concurrency; the advent of PROLOG systems for concurrent applications (see for example [22, 92]), especially operating systems, and the increasing interest in logic programming as a language for interactive systems makes the specification of a semantics of infinite behaviors a very up to date problem.

Henceforth, the second solution seems more reasonable. Simultaneously handling both recursion and negation is a very hard job. Difficulties arise mainly from two contrasting features of infinite computations and negation causing that any (effective) model of the former needs a continuous (and then monotonic) fixpoint operator which is forbidden by the intrinsic non-monotonic nature of negation.

However, results obtained from a separated treatment of the two aspects of negation and infinite computations are not unrelated, and solving the problem of infinite computations would help also in improving the elegance of the theory of negation. In fact, as we will see in Section 3.3, negation introduces an asymmetry in the fixpoint semantics of the logic programs that could be eliminated by introducing in the semantics some infinite objects.

The third solution arose from the idea that the two truth values *true* and *false* are not sufficient to represent all the situations that a program could bring forth by its execution, since it may either succeed or fail or go on forever giving no answers. The last possibility does not have a corresponding truth value in classical logic, whereas it can be represented in the three-valued logic by the undefined (or unknown) truth value, u. Introduced by Kleene to deal with partial recursive functions and predicates, three-valued logic provides a great advantage when used in connection with logic programming, namely it avoids the above mentioned difficulty related to the non-monotonicity of the fixpoint operator in presence of negation. This methods has been developed by Fitting ([41]) and Kunen ([66, 67]).

1.1 Contributions of the thesis

This thesis is about the non-ground semantics of logic programs and its extensions to general logic programs. The main contributions can be resumed as follows.

Positive programs

The main results are related to the C-semantics, ([39]), and consists in

- The introduction of a new negation rule which allows to infer existentially closed negated atoms. These represent the failure set of the C-semantics.
- The characterization of infinite computations for a language subsuming logic programming language, by means of a denotational-like approach. When restricted to the terminating successful computations this model corresponds to the C-semantics.
- The introduction of a notion of negation (again within the C-semantics) able to take into account also infinite computations, and its characterization by using the Heyting algebras.

Some hints and proposals about analogous characterizations for the S-semantics are also sketched as a basis for future work.

General programs

An interpreter for general programs based on the new negation rule is presented, which is orthogonal to the SLDNF-resolution. The amalgamation of the new rule and the NAF rule produces a powerful system which is able to deal with logic programs with any first-order formulas in the body of the clauses.

1.2 Summary

The rest of the thesis is organized as follows.

Chapter 2 is devoted to some backgrounds on the semantics of logic programs.

In Chapter 3 we introduce the NAI rule, a new rule for inferring negative information from positive programs, and we characterize the corresponding failure set. Finally, we summarize the situation in the light of the new results.

In Chapter 4 we present the Γ_clp language which generalizes (constraint) logic programming language and we define a model for it, which is able to capture both the finite and the infinite results of the computations of a goal from a given program.

In Chapter 5 we extend the model developed in Chapter 4 to negative goals. The resulting notion of negation is shown to be the intuitionistic negation.

Chapter 6 is centered on general programs. The rule introduced in Chapter 3 is used to define three kinds of interpreter for these programs: the SLDNI-resolution, in which negative subgoals are treated by the NAI rule; the SLDNFI-resolution, which is based on an amalgamation of the NAI and the NAF rules; the NAIF-resolution in which the rule for negation is a slight modification of the amalgamation rule aimed at improving the performance of the interpreter.

Finally, in Chapter 7 we sum up the thesis and we discuss the most promising directions for further research.

Chapter 2

Logic programs

In which we introduce basic notations and results about logic programs.

In its original form, ([65]), logic programming is a programming language consisting of pure Horn clauses logic (a small fragment of first order logic) with the computational mechanism called SLD-resolution (a specific form of theorem proving based on the resolution principle[1]). Programs written in this language are usually called definite, or positive programs. Starting from the interpretation of predicate logic as a programming language of [65], Van Emden and Kowalski ([105]) provided (positive) logic programs with a declarative semantics which is nowadays largely agreed upon, and can be safely considered as *the* semantics of logic programs. By introducing the transformation T_P associated with a given program P (successively called by Clark ([20]) the *immediate consequence operator*), Van Emden and Kowalski defined a fixpoint semantics for logic programs similar to that for more conventional programming languages (see for instance [25]), In addition, they introduced a model-theoretic characterization of logic programs, the *least Herbrand model*, closely related to the underlying logic language and, in general not used in case of other programming languages.

Even if the results about the semantics of logic programs are nowadays well-known in the field, it is worth while to recall them as well as the basic notations and terminology used in logic programming.

2.1 Background

A language consists of three disjoint sets: a set Con of *function symbols*, a set $Pred$ of *predicate symbols* and a set Var of *variable symbols*. Each symbol in Con and $Pred$ is associated with a number representing its arity.

Let $Term$ be the set of terms $t, u \ldots$ built on Con and Var. The Herbrand universe U is the set of all ground terms. A substitution is a mapping $\theta : Var \to Term$ such that the set $Dom(\theta) = \{x \mid \theta(x) \neq x\}$ (*domain* of θ) is finite. The *codomain* of θ is the set $Cod(\theta) = \{t \mid t \in Term \text{ and } \exists x \in Dom(\theta) \text{ such that } t = \theta(x)\}$. For a set of variables V, the *restriction of* θ *to* V, denoted by $\theta_{|_V}$, is the substitution defined by $\theta_{|_V}(x) = \theta(x)$ if $x \notin V$, and $\theta_{|_V}(x) = x$ otherwise. The empty substitution is denoted by ε. The notion of substitution can be naturally extended to terms. The composition $\theta\sigma$ of the substitutions θ and σ is defined as the functional composition. The pre-ordering $\leq$ on substitutions is such that $\theta \leq \sigma$ iff there exists θ' such that $\theta\theta' = \sigma$.

[1]The interested reader can find useful references to the literature related to this subject in the Lloyd's book [71]

The set *Atom* is a set of objects $A, B \ldots$ of the form $p(t_1, \ldots, t_n)$, where p is a predicate with arity n, and $\forall i \in [1, n], \quad t_i \in \mathit{Term}$. A *literal* is either an atom or the negation of an atom. The Herbrand Base $\mathcal{B}$ is the set of all ground atoms. The application of the substitution θ to the atom A is denoted by $A\theta$. The relation $A \leq A'$ (A is more general than A') holds iff there exists θ such that $A\theta = A'$. The relation $\leq$ is a preorder, and the associated equivalence relation (*renaming*) will be denoted by $\approx$. In other words, A is a renaming of A' if there exist θ and θ' such that $A\theta = A'$, and $A'\theta' = A$; θ (θ') is called a renaming for A (A').

We use $\forall A$, $\exists A$ to denote respectively $\forall x_1, \ldots, \forall x_n.A$ and $\exists x_1, \ldots, \exists x_n.A$, where $x_1, \ldots, x_n$ are all the variables occurring in A.

Two atoms A and A' are *unifiable* iff there exists a substitution θ such that $A\theta = A'\theta$. Such substitution is called *unifier* of A and A'. If A and A' are unifiable, then there exists a smallest unifier w.r.t. $\leq$, apart from renaming, which is called the *most general unifier* (mgu) of A and A', and denoted by $mgu(A, A')$. The notion of mgu generalizes to sequences of atoms or terms in the obvious way.

A *definite clause* is a formula $H \leftarrow B_1, \ldots, B_n$ ($n \geq 0$), where H and the B_i's are atoms, "$\leftarrow$" and "," denote logic implication and conjunction respectively, and all variables are universally quantified. H is the *head* of the clause and $B_1, \ldots, B_n$ is the *body*. If the body is empty then the clause is a *unit clause*. A *definite Horn program*, (or simply *program*), is a finite set of definite clauses $P = \{C_1, \ldots, C_n\}$. A *goal* G is a formula $\leftarrow A_1, \ldots, A_m$, where each A_i is an atom. We will often refer to them as *positive* or *definite* programs and goals.

Given a goal G of the form $\leftarrow A_1, \ldots, A_m$, and a program P, a *derivation step* $G \stackrel{\theta}{\longmapsto} G'$ is possible if there exists a clause $H \leftarrow B_1, \ldots B_n$ which is obtained by renaming in an appropriate way (see later) one of the clauses in P, such that $\theta = mgu(A_i, H)$. The resulting goal G' is $\leftarrow (A_1, \ldots, A_{i_1}, B_1, \ldots B_n, A_{i+1}, \ldots, A_m)\theta$. The atom A_i is called *selected atom*. Given a particular *selection rule*, namely a rule which associates to each (occurrence of a) goal a selected atom, an *SLD-derivation* for $P \cup \{G\}$ is a sequence of derivation steps $G = G_0 \stackrel{\theta_0}{\longmapsto} G_1 \stackrel{\theta_1}{\longmapsto} \ldots \stackrel{\theta_{n-1}}{\longmapsto} G_n$, with $n \geq 0$. In short $G \stackrel{\theta}{\longmapsto}^* G_n$ where $\theta = (\theta_0\theta_1 \ldots \theta_{n-1})_{|\bar{x}}$ if $n \geq 1$, $\theta = \epsilon$ if $n = 0$; here $\bar{x}$ represent the variables of G. The renaming of a clause used in a derivation has to be done in such a way that it does not contain variables in common with any goal occurring in the prefix of the derivation until the use of that clause (*standardization apart*). Without loss of generality, we will assume that each mgu μ generated in a derivation is *idempotent* (i.e. $\mu\mu = \mu$) and *relevant* (i.e. μ affects only variables which occur in the atoms to be unified). If $G \stackrel{\theta}{\longmapsto}^* \Box$, where $\Box$ is the *empty goal*, then the derivation is a *SLD-refutation*, and θ is the corresponding *computed answer substitution* (c.a.s.). An SLD-derivation is *failed* if it is finite, maximal (i.e. it is not the prefix of a longer derivation) and it is not a refutation. A derivation is *fair* if it is either finite or every atom which occurs in it will be eventually selected.

An SLD-tree for $P \cup \{G\}$ is a way of representing all derivations for G in P via a fixed selection rule R. It is a tree in which

(i) the nodes are goals, and the root is G,

(ii) the children of a node are all the goals obtained by performing a resolution step with a renamed clause of P whose head is unifiable with the atom selected according to R.

Note that a branch of an SLD-tree corresponds to an SLD derivation. An SLD-tree is *successful* if it contains the empty clause, and it is *finitely failed* if it is finite and not successful. It is fair if all its branches are fair.

2.1.1 Ground semantics

We refer to [71, 4] for the standard notions of Herbrand base B, Herbrand interpretations $I, J, \ldots$ and the immediate consequence operator T_P associated to a program P.

The *ground success set* is defined to be the set

$$SS = \{A \mid A \text{ is ground, and } \leftarrow A \longmapsto^* \Box\}.$$

The basic results of the standard semantics of positive programs are expressed by the following characterization of SS. Let P be a program and A a ground atom. Then we have:

$$\begin{aligned} SS &= \{A \mid A \text{ is ground and } P \models A\} \\ &= M_P \text{ (the least Herbrand model of) } P \\ &= T_P \uparrow \omega. \end{aligned} \tag{2.1}$$

In the following we refer to the above characterization as the *ground success set semantics*.

2.1.2 Non-ground semantics

Standard semantics does not reflect some more "procedural" aspects of the correspondence between the operational and declarative semantics of programs. The reason is that it considers only ground atoms, thereby hiding the ability of a program to compute substitutions. On the other hand, this concept is a fundamental one, when we look at Horn clauses logic specifically as a programming language. From this point of view, it doesn't matter the general deductive power of the theorem prover; what we are interested in are the substitutions produced during the resolution process by unification, besides knowing if a formula is a theorem of the given program (logical theory). In [71] the correspondence between operational and declarative semantics is formulated in terms of two notions: "computed answer substitution" and "correct answer substitution". The latter represents a way to declaratively capture the bindings produced during computation to the variables occurring in a goal, which are operationally expressed by the former. However, correct answer substitutions does not yet fully characterize computed answer substitutions, as they may be instances of the substitutions which are actually computed.

The problem of declaratively characterizing the operational behavior of programs has been studied in the papers [39, 40], where a more radical solution has been proposed based on the idea of allowing variables in the interpretation structures. Two non-ground semantics have been introduced which differ from each other in dealing with the presence of variables in the domain of interpretation. One follows an approach firstly proposed (for different purposes) in [20] and considers interpretations to be *upward closed* subsets of atoms. It is called the C-semantics. The other considers just sets of atoms, but then it must to refer to a new notion of truth (the *S-truth*) for defining the meaning of formulas. It is called the S-semantics. It turns out that S-semantics is a powerful semantics which has a stronger relation with the operational semantics than the former. In fact, it characterizes exactly the set of computed answer substitutions. Starting from the semantics introduced in the above mentioned papers, it has been developed a general approach to the semantics of positive programs, whose aim is modeling the observables behaviors, possibly in a compositional way, for a variety of logic languages, ranging from positive logic programs, to general logic programs, constraint logic programs, concurrent constraint programs and languages with inheritance mechanism. The approach is based on the idea of using extended interpretations (called π-interpretations) which are more expressive than usual (Herbrand) interpretations and therefore more appropriate to model observable behaviors of computations. An exhaustive description and useful references to this approach can be found in [43].

In this thesis we will refer to the non-ground semantics as developed in the original papers [39, 40]. We recall the main results. They will be used as well as the ground success semantics to establish in the next chapter some "duality" with the theory of negative information.

Following [39], we summarize the notion of Herbrand structure enriched with variables, and the corresponding immediate consequence operators respectively for the C-semantics and the S-semantics.

The "Herbrand base with variables" is the set of equivalence classes (quotient set) of the set of all atoms with respect to the equivalence relation $\approx$:

$$\mathcal{B}_{\mathrm{v}} = Atom_{/\approx}.$$

For the sake of simplicity, we denote the equivalence class of an atom with the atom itself.

We introduce the following operators on subsets I_{v} of $\mathcal{B}_{\mathrm{v}}$, which will be useful in the following.

- $Down(I_{\mathrm{v}}) = \{A \mid \text{there exists } A' \in I_{\mathrm{v}} \text{ s.t. } A \leq A'\}$
- $Ground(I_{\mathrm{v}}) = \{A \mid A \in I_{\mathrm{v}} \text{ and } A \text{ is ground}\}$
- $Up(I_{\mathrm{v}}) = \{A \mid \text{there exists } A' \in I_{\mathrm{v}} \text{ s.t. } A' \leq A\}$.

I_{v} is upward closed if $I_{\mathrm{v}} = Up(I_{\mathrm{v}})$.

The immediate consequence operator of the C-semantics, T_C, on Herbrand interpretations with variables is

$$T_C(I_{\mathrm{v}}) = \{A \mid \begin{array}{l} \text{there exist } H \leftarrow \bar{B} \in P_{\mathrm{ren}} \text{ and} \\ \text{a substitution } \theta \text{ s.t.} \\ \bar{B}\theta \in I_{\mathrm{v}} \text{ and } A = H\theta\}. \end{array}$$

where P_{ren} indicates the closure of P under renaming and $\bar{B}$ denotes a sequence of atoms.

The set of non-ground atoms which can be refuted with an empty computed answer substitution (called here *NESS* for Non-ground Empty computed answer substitutions Success Set) is defined by

$$NESS = \{A \mid \leftarrow A \overset{\varepsilon}{\longmapsto}{}^{*} \Box\}.$$

We recall the model-theoretic and fixpoint characterization of *NESS*, which extends the results in (2.1).

$$\begin{array}{rcl} NESS & = & \{A \mid P \models \forall A\} \\ & = & M_v \text{ (the least Herbrand model with variables)} \\ & = & T_C \uparrow \omega. \end{array}$$

The immediate consequence operator of the S-semantics, T_S, on Herbrand interpretations with variables is

$$T_S(I_{\mathrm{v}}) = \{A \mid \begin{array}{l} \text{there exists } H \leftarrow B_1, \ldots, B_n \in P, \\ \text{there exist } B'_1, \ldots, B'_n \text{ variants of atoms in } I_{\mathrm{v}} \\ \text{ and renamed apart,} \\ \text{there exists } \theta = mgu((B_1, \ldots, B_n), (B'_1, \ldots, B'_n)) \\ \text{ and } A = H\vartheta\}. \end{array}$$

The set of computed answer substitutions (called here CSS for Computed answer substitutions Success Set) is defined by:

$$\begin{aligned} CSS = \{A \mid\ & \text{there exists } A' \text{ and } \vartheta, \\ & \text{such that } A' \text{ is elementary}, \\ & A' \stackrel{\vartheta}{\longmapsto} \Box,\ A = A'\vartheta\}, \end{aligned}$$

where the atom A' being elementary means that there exist a predicate symbol $p \in Pred$ and distinct variables $x_1, \ldots, x_n$ in Var such that $A' = p(x_1, \ldots, x_n)$.

Also for CSS it is shown in [39] its equivalence with the least fixpoint of T_S and to the least Herbrand S-model (i.e. the least Herbrand model under the notion of S-truth), M_S. Thus we have:

$$\begin{aligned} CSS &= M_S \\ &= T_S \uparrow \omega. \end{aligned}$$

The relationship between the C-semantic and the S-semantic is given by the following proposition.

Proposition 2.1.1 *[39] Let I be an interpretation. Then*

$$T_C(I) = Up(T_S(I)).$$

Chapter 3

Negative information: the NAI rule

In which we consider positive programs and the ways to infer negative information from them. As well as positive behaviors of programs are observed by success sets, negative behaviors determine other observables: the failure sets.

The most naïve way to obtain some negative information from a logic program would be to put a query containing some negative literals. This is wholly legal, but we must not expect to obtain any answer other than "no". In fact, if the program consists entirely of definite Horn clauses, then SLD-resolution is complete for classical negation, because a query containing a negative literal cannot succeed, according to the fact that it cannot be a logical consequence of the program (this has a model, the Herbrand base, in which all atoms are true).

This means that the only information we can deduce from a positive program consists of positive facts, namely all the atoms which are "logically" deducible from the program. The intended meaning of "logically" is "in the classical logic"; thus we can expect to obtain something more by using some (non-classical) inference rule. Of course, any negative information we will be able to infer, it will not be the classical negation. From a practical point of view this is not a problem. On the contrary, almost all applications of logic programming prefer other forms of negation which are different from the classical one. For example, in the context of deductive databases (but not only here) negative facts are deduced from a positive program (the database) by using what McCarthy calls a *common-sense reasoning*. The idea is applying defaults for inferring information that cannot be proved. One common such default is *negation as failure*: essentially if there is no reason to infer an atomic sentence to be true, infer it to be false. Classical logic does not allow such a reasoning. But, there is also a much more practical reason that justifies the use of inference rules instead of classical negation. There are circumstances where it is relatively easy to specify when a relation is true but much harder to specify directly when it is false. Consider the definition of the word "ancestor": it is relatively easy to specify recursively when x is an ancestor of y, but there is no such straightforward recursive definition of "not an ancestor". In such cases it seems desirable to use the inference engine to specify only when the relation is true, the easy part, and to leave the hard part to the automated system.

Even if the above arguments speak in favor of the use of inference rules for the treatment of negation, one can believe that a more natural solution should be pursued, namely to permit the full use of classical negation in the queries and to define an automatic theorem prover which is sound and complete for the full first order logic. What is discouraging in this solution is the complexity of the resulting procedure which cannot be compared with the very efficient SLD-resolution. In spite of this, some authors have envisaged this possibility and have introduced systems admitting explicit negation. Examples of these are the Negation as Inconsistency ([42]), the Naive Negation of [87], and an implementation of Prolog described in [83].

Inference rules (or metarules as someone prefers to call these rules) are formulated by referring to a notion of provability which usually is that of first-order logic. For our purposes it is sufficient to know that it is equivalent here to provability by means of SLD-resolution.

In this chapter we dwell upon the concept of *failure*, a fundamental concept in the treatment of negation in logic programming. Even if not explicitly specified, we always refer to positive programs. Given a program P we consider the sets of atoms collecting some negative information that can be drawn from P by using a negation rule. Each of them represents a *failure set* for the program P. In line with the approach of [40], we regard failure sets as some *observable properties* of a program, and we look at the problem of providing them of equivalent operational, fixpoint and model-theoretic characterizations.

After recalling the two failure sets based on the Negation As (finite) Failure (NAF) rule (i.e. the classical ground finite failure sets and the non-ground finite failure set), we present the notion of "failure by finite instantiation", for which a failed SLD-tree is not necessarily finite. The associated failure set is shown to be formed by those atoms whose negation can be inferred by a new rule: the Negation As (finite) Instantiation (NAI) rule. The meaning of these negated atoms is their existential closure. Thus, NAI rule allows us to infer more negative information than NAF rule, since the latter is able to infer only universally closed negated atoms.

3.1 Negation As Failure

The "definition" of negation as failure given above is quite vague. For example, if we assume the formula

$$\mathit{Bird(Tweety)} \vee \mathit{Bird(Opus)}$$

to be true, then a simple-minded negation as failure would lead one to infer $\neg \mathit{Bird(Tweety)}$, as the given formula does not imply *Bird(Tweety)*. Similarly, one would infer $\neg \mathit{Bird(Opus)}$. But inferring both of them contradicts the original assumption. We need a more precise statement of negation as failure. Actually, due to the variety of intuitions leading to this notion, there is no a unique definition of negation as failure. As a proof of this fact consider the proliferation of semantics which have been proposed in the context of general programs (see Section 6.4). For positive programs, the presence of a well-settled declarative semantics has deeply affected the intuition behind negation as failure. Here, the truth of a ground atom A is determined by checking its belonging to the least Herbrand model M_P of the program P; thus determining its falsity results in checking that A does not belong to M_P. The resulting notion of negation as failure is very simple: if A cannot be proved[1] from P, then infer $\neg A$. This is usually called the *closed world assumption* rule (cwa) and it was first considered in [86]. However, when we try to transpose this intuition from the declarative level to the operational one, we soon realize that there is no way to make it effective, since the complement of M_P is not recursively enumerable. The alternative is a more procedural reading of the programs. This can be obtained, for example, by *observing* the failure of the SLD-resolution instead of deducing it from the absence of a success. By noting that observability implies that a *finite* amount of time has to be considered, the resulting notion is that of *negation as finite failure* (NAF): if an attempt to prove a ground atom A using SLD-resolution fails finitely, then infer $\neg A$, ([23]). The concept of *finite failure* becomes then a central one.

[1] The two interpretations "proved in the first-order logic", and "proved by SLD-resolution" are equivalent.

3.1.1 Ground finite failure

The *ground finite failure* set, *FF*, of a program P is the set of all ground atoms A such that there exists a finitely failed SLD-tree, i.e. an SLD-tree which is finite and contains no empty clause, with $\leftarrow A$ as root. The following theorem, given in [4], collects two characterizations of *FF* due to respectively Apt and van Emden ([7]), and Lassez and Maher ([69]).

Theorem 3.1.1 *Let P be a program and A a ground atom. Then the following are equivalent:*

a) $A \in FF$

b) $A \notin T_P{\downarrow}\omega$

c) *Every fair SLD-tree for $P \cup \{\leftarrow A\}$ is finitely failed.*

The model-theoretic characterization of *FF* derives from two results which are known as correctness and completeness of NAF and have been shown respectively by Clark ([23]) and by Wolfram, Maher and Lassez ([110]). They mean that the NAF rule infers only and all the ground atoms (or conjunction of atoms) which are true in every model of the *completion*, $Comp(P)$, of a given program P. The theory of completion is due to Clark ([23]), which proposed it as the reference theory for establishing the truth of negative formulas, since they cannot be logical consequences of a positive program. It represents the standard semantics of logic programs with NAF, also called the Clark completion semantics. The construction of the completion of P proceeds as follows. First construct the set of the *completed definitions* of the predicates in P, $IFF(P)$, i.e. the set of formulas

$$p(\bar{x}) \leftrightarrow \bigvee_{i \in [1,s]} E_i,$$

where $\bar{x} = x_1, \ldots, x_n$ are variables not appearing anywhere in the definition of p, and, for all i' s, if the i-th clause in P with p in the head is of the form $p(\bar{t_i}) \leftarrow L_1, \ldots, L_{n_i}$, then E_i is the formula $\exists \bar{y_i}.(\bar{x} = \bar{t_i} \wedge L_1, \ldots, L_{n_i})$, where $\bar{y_i}$ denotes the variables of the original clause.

Then, form $Comp(P)$ as $IFF(P) \cup CET$, where CET (Clark's Equational Theory) is a set of axioms forcing the interpretation of "=" to be the identity on the Herbrand universe. For a discussion of these axioms see, for instance, [95], where they are partitioned in two sets, respectively called equality axioms (EA) and freeness axioms (FA).

The standard semantics of the finite failure is given by the following equations expressing the equivalence of the operational, model-theoretic and fixpoint characterizations of *FF*.

$$\begin{aligned} FF &= \{A \mid A \text{is ground and there exists a finitely} \\ &\qquad\quad \text{failed SLD-tree for } P \cup \{\leftarrow A\}\ \} \\ &= \{A \mid Comp(P) \models \neg A\} \\ &= \mathcal{B} \setminus T_P{\downarrow}\omega. \end{aligned} \tag{3.1}$$

3.1.2 Non-Ground Finite Failure

The introduction in [39] of the two non-ground success sets *NESS* and *CSS*, put the natural question of characterizing some non-ground finite failure sets playing the same role as *FF* w.r.t. the ground success set *SS*, i.e. that of being *dual* sets. In fact, as already mentioned, from a computational point of view the set-theoretical complement of the ground success set is of scarce interest, whereas more significant is the set representing its "semantic" complement (or dual).

Similar considerations hold for the non-ground case too. In [70] the set of non-ground atoms which have a finitely failed computation, *NGFF*, has been studied. However, it represents the exact counterpart of the set *NGSS* (Non-Ground Success Set), i.e. the set of those atoms which have a successful computation with any computed answer substitution, which is different from both *NESS* and *CSS*. By exploiting some of the results for (the T_C operator of) the C-semantics developed in [39] *NGSS* and *NGFF* have been provided of a fixpoint characterization showing some analogies with the fixpoint characterization of the corresponding ground sets. In order to obtain such a characterization, an extension of the T_C operator on "complete" (in a topological sense) Herbrand interpretations (with variables) has been necessary to get a greatest fixpoint. It is shown that:

$$NGFF = \mathcal{B}_{\mathrm{v}} \setminus \mho(gfp(T'_C)),$$

and

$$NGSS = \mho(lfp(T'_C)),$$

where $\mho$ is the function transforming a complete interpretation into a subset of $\mathcal{B}_{\mathrm{v}}$ and T'_C stands for the extended T_C.

3.2 Negation As Instantiation

There are at least two reasons why the above approach is not completely satisfactory. One is that the notion of (ground or non-ground) finite failure is somehow restrictive, since by it we can infer only a small part of the negative information which could be drawn from a program, namely the universally quantified negative literals; the other is that we would like to define generalizations of both the C-semantics and the S-semantics modeling finite failures. In this section we show how the situation can be improved relatively to the first limitation. As a byproduct we obtain the characterization of a failure set for the C-semantics which is the negative counterpart of *NESS*. Unfortunately, concerning the S-semantics, a model for the failure does not yet exists. In Section 3.3 we will give some insights of this. Now, we present a new rule to infer existentially closed negative literals and we characterize such formulas declaratively, by using the C-semantics.

Let's illustrate the basic idea. If all derivations for $\leftarrow A$ in a program P either fail or instantiate some of the variables of A, then $\forall A$ is not a logical consequence of the Clark's completion of P, $Comp(P)$. Therefore it is consistent to infer $\neg\forall A$, namely $\exists\neg A$. We want to extend now the reference theory ($Comp(P)$) in order to *validly* infer $\exists\neg A$. To this purpose, note that if every branch of the SLD-tree for $P \cup \{\leftarrow A\}$ either fails or instantiates some of the variables of A, then for a ground substitution η instantiating all variables to distinct fresh constants, the SLD-tree for $P \cup \{\leftarrow A\eta\}$ finitely fails. Thus, by the soundness of NAF, we can deduce $\neg A\eta$, and finally $\exists\neg A$. Therefore, to obtain the appropriate reference theory, it is sufficient to extend the underlying language by infinitely many constant symbols, and consider the completion of P over the extended language L, $Comp_L(P)$. Based on this concept of *finite instantiation* we define a new negation rule which we call *Negation As Instantiation* (NAI) and an operational semantics for negation, the Failure by Finite Instantiation set (the *FFI* set), consisting of the atoms whose existentially quantified negation can be inferred.

The fixpoint characterization of the *FFI* set is obtained by using the C-semantics, and, as we will see, this leads to a precise analogy with the standard semantics.

3.2.1 Failure by instantiation

In this section we present a new notion of failure and, correspondingly, a new (non-ground) failure set to be interpreted as negative information. For this set we give a characterization similar to (3.1), using the T_C operator. Finally we show that this set contains exactly the atoms A for which $Comp_L(P) \models \exists\neg A$, holds, where $Comp_L(P)$ is the completion of P with respect to the reference language L (containing infinite constant symbols). This set represents the actual dual set of *NESS*.

The operator *var* returns the set of variables occurring in an expression (term, atom, goal,...). For a substitution θ and a set of variables V, the property $Inst(\theta, V)$ holds iff θ strictly instantiates the variables of V.

Definition 3.2.1 *Let $V = \{x_1, \ldots, x_n\}$, θ be a substitution and p a predicate symbol of arity n.*

$$Inst(\theta, V) \Leftrightarrow p(x_1, \ldots, x_n)\theta \not\leq p(x_1, \ldots, x_n).$$

Definition 3.2.2 (Finitely Instantiating SLD-tree) *Let P be a program, G a goal and $V \subseteq var(G)$. Let TR be an SLD-tree for $P \cup \{G\}$. Then*

1. *TR instantiates V at level k iff for every branch ξ of TR one of the following holds:*
   - *ξ fails at level*[2] *$k' \leq k$, or*
   - *$Inst(\theta_1 \cdots \theta_{k'}, V)$ is true, where $\theta_1, \ldots, \theta_{k'}$ are the substitutions labeling the edges of ξ up to level $k' \leq k$.*
2. *TR is a finitely instantiating SLD-tree for $P \cup \{G\}$ iff $\exists k \geq 1$ s.t. TR instantiates $var(G)$ at level k.*

Definition 3.2.3 (Failure by Finite Instantiation) *Let P be a program. The failure by finite instantiation set of P is*

$$FFI = \{A \in Atom \mid \text{there exists a finitely instantiating SLD-tree for } P \cup \{\leftarrow A\}\}.$$

In the following we will show that *FFI* corresponds to the set of atoms not belonging to $T_C{\downarrow}\omega$.

3.2.2 The NAI rule

We present here a new rule for inferring negative information based on the concept of finite instantiation. We call this rule *Negation As Instantiation* (NAI) and we define it as:

$$\frac{A \in FFI}{\exists\neg A}$$

The NAI rule subsumes the NAF rule in the sense that for ground atoms it coincides with the NAF rule, and furthermore it has the advantage of an efficient implementation. To show that $\exists\neg A$ holds we perform an exhaustive search for a proof of A. If every possible proof fails or instantiates some variables of A (that is if it is possible to construct a finitely instantiating SLD-tree for $P \cup \{\leftarrow A\}$), then $\exists\neg A$ is inferred.

[2]By level of ξ we mean the number of nodes in ξ.

Examples

We show some examples to clarify the possible use of the NAI rule. We consider here positive programs and goals consisting of only one existentially closed negative literal.

Example 3.2.4 *Consider the following programs:*

$$P_1 = \{\ p(a) \leftarrow, \quad r(b) \leftarrow \quad \}$$

$$P_2 = \{\ p(x) \leftarrow, \quad r(b) \leftarrow \quad \}.$$

We have:

$$Comp(P_1) \models \neg p(b), \text{ hence } Comp(P_1) \models \exists\neg p(x),$$

and

$$Comp(P_2) \models \forall p(x), \text{ hence } Comp(P_2) \models \neg\exists\neg p(x).$$

There exists a finitely instantiating SLD-tree for $P_1 \cup \{\leftarrow p(x)\}$*, but not for* $P_2 \cup \{\leftarrow p(x)\}$*. Hence the NAI rule will allow us to derive* $\exists\neg p(x)$ *in* P_1*, and not in* P_2*. Note that neither* $P_1 \cup \{\leftarrow p(x)\}$ *nor* $P_2 \cup \{\leftarrow p(x)\}$ *have a finitely failed SLD-tree, so no conclusion can be drawn by the NAF rule (even when extended to deal with non-ground negative literals as it is done in [4], Section 6). This is in accordance with the fact that* $Comp(P_i) \not\models \forall\neg p(x)$ *for both* $i = 1, 2$*.*

Example 3.2.5 *The following program defines a predicate whose third argument is the sum of the others.*

$$P = \{\ plus(x, 0, x) \leftarrow, \quad plus(x, s(y), s(z)) \leftarrow plus(x, y, z) \quad \}$$

We have, for instance, that $Comp(P) \models \exists\neg plus(x, s^2(0), y)$*. From the NAI rule, we can derive* $\exists\neg plus(x, s^2(0), y)$*; in fact,* $P \cup \{\leftarrow plus(x, s^2(0), y)\}$ *has a finitely instantiating SLD-tree where the goal variable* y *results instantiated to* $s^2(x)$*. On the other hand, we have that for all* $n \in \omega$

$$Comp(P) \models \neg\exists\neg plus(x, s^n(0), s^n(x))$$

holds, and the NAI rule fails, since $P \cup \{\leftarrow plus(x, s^n(0), s^n(x))$ *has a non-instantiating SLD-refutation.*

Let's now consider an example of application of the NAI rule in the case of a finitely instantiating, but infinite, SLD-tree.

Example 3.2.6 *Consider the following program*

$$P = \{\ p(a) \leftarrow q(x), \quad q(x) \leftarrow q(x) \quad \}$$

We have that the SLD-tree for $P \cup \{\leftarrow p(y)\}$ *is infinite, but finitely instantiating. In fact* y *is instantiated to* a *already at the second level. Hence the NAI rule allows us to derive* $\exists\neg p(y)$*, while no conclusion can be derived from the NAF rule, in accordance with the fact that* $Comp(P) \not\models \forall\neg p(y)$*.*

On the other hand, the SLD-tree for $P \cup \{\leftarrow q(y)\}$ *is infinite, but not finitely instantiating. Hence no conclusion for* q *can be derived from either the NAI or the NAF rule.*

In these examples it is not clear the relation between the NAI rule and the completion of a program; and more in general it is not clear what is the semantical justification of such a rule. We will investigate this aspect in Section 3.2.5, whereas now we will focus on a characterization of *FFI* in terms of the immediate consequence operator T_C.

3.2.3 Fixpoint characterization of failure by instantiation

We show that the set *FFI* enjoys a characterization similar to that of finite failure as given by Theorem 3.1.1. In fact, there is a close correspondence between failure by instantiation and finite failure, as we show in this section. The basic idea is that a finitely instantiating tree becomes a finitely failed tree when the variables of the initial goal are replaced by constant symbols not occurring in the program. In fact if the unification of A and B requires to instantiate the variables of A, then no unification is possible if we replace the variables of A by constant symbols not occurring in B. In order to exploit this relation in the proofs, technically we have to consider Herbrand structures, T_P, *FF* etc. defined on a language extended with the additional constant symbols.

Given an expression E, a sequence of variables $\bar{x}$ and a sequence of terms $\bar{t}$ of the same length, $E[\bar{t}/\bar{x}]$ denotes the expression obtained from E by replacing verbatim each variable of $\bar{x}$ by the corresponding term of $\bar{t}$. Furthermore, $\theta[\bar{t}/\bar{x}]$ is the substitution defined by $\theta[\bar{t}/\bar{x}](y) = \theta(y)[\bar{t}/\bar{x}]$.

Proposition 3.2.7 *Let $A \in Atom$, let $\bar{x}$ be the sequence of all variables occurring in A, and let $\bar{d}$ be a sequence of fresh constant symbols. Assume $A \in FFI$. Then $A[\bar{d}/\bar{x}] \in FF$.*

Proof Note that if $P \cup \{\leftarrow A[\bar{d}/\bar{x}]\}$ has a non-failed SLD-derivation, then we obtain a non-failed SLD-derivation for $P \cup \{\leftarrow A\}$ by replacing verbatim the occurrences of symbols in $\bar{d}$ by the corresponding symbols in $\bar{x}$ in the mgu's and in the goals. The resulting mgu's do not bind $\bar{x}$, hence the derivation does not instantiate $\bar{x}$. Therefore,

$$
\begin{array}{lll}
A[\bar{d}/\bar{x}] \notin FF & \Rightarrow & \text{every SLD-tree } TR' \text{ for } P \cup \{\leftarrow A[\bar{d}/\bar{x}]\} \\
 & & \text{has a non-failed branch} \\
 & & \\
 & \Rightarrow & \text{every SLD-tree } TR \text{ for } P \cup \{\leftarrow A(\bar{x})\} \\
 & & \text{has a non-failed branch with mgu's } \theta_1, \theta_2, \ldots \\
 & & \text{such that } \forall n. \neg Inst(\theta_1 \cdots \theta_n, \bar{x}) \\
 & & \\
 & \Rightarrow & A \notin FFI.
\end{array}
$$

□

Let us recall the relation between the operators T_P and T_C given in [70].

Lemma 3.2.8 *For every n, $T{\downarrow}n = Ground(T_C{\downarrow}n)$ holds.*

Lemma 3.2.9 $T_P{\downarrow}\omega = Ground(T_C{\downarrow}\omega)$.

Proof

$$
\begin{array}{ll}
& A \in T\!\downarrow\!\omega \\
\Leftrightarrow & \\
& A \in \cap_{n<\omega} T\!\downarrow\! n \\
\Leftrightarrow & \\
& \forall n \quad A \in T\!\downarrow\! n \\
\Leftrightarrow & \quad \{\text{Lemma 3.2.8}\} \\
& \forall n \quad A \in \mathit{Ground}(T_C\!\downarrow\! n) \\
\Leftrightarrow & \quad \{\text{upward closedness of } T_C\!\downarrow\! n\text{, see Definition 6.2 in [39]}\} \\
& A \text{ is ground and } \forall n. A \in T_C\!\downarrow\! n \\
\Leftrightarrow & \\
& A \text{ is ground and } A \in T_C\!\downarrow\!\omega \\
\Leftrightarrow & \\
& A \in \mathit{Ground}(T_C\!\downarrow\!\omega).
\end{array}
$$

□

Proposition 3.2.10 $A \in \mathit{FFI} \Rightarrow A \notin T_C\!\downarrow\!\omega$.

Proof Let $\bar{d}$ be a sequence of new constant symbols. We have:

$$
\begin{array}{ll}
& A \in \mathit{FFI} \\
\Rightarrow & \quad \{\text{Proposition 3.2.7}\} \\
& A[\bar{d}/\bar{x}] \in \mathit{FF} \\
\Rightarrow & \quad \{\text{Theorem 3.1.1}\} \\
& A[\bar{d}/\bar{x}] \notin T_P\!\downarrow\!\omega \\
\Rightarrow & \quad \{\text{Lemma 3.2.9}\} \\
& A[\bar{d}/\bar{x}] \notin \mathit{Ground}(T_C\!\downarrow\!\omega) \\
\Rightarrow & \quad \{\text{upward closeness of } T_C\!\downarrow\!\omega\} \\
& A(\bar{x}) \notin T_C\!\downarrow\!\omega.
\end{array}
$$

□

Observe that the definition of FFI is existential, in the sense that $A \in \mathit{FFI}$ iff there exists at least a selection rule which gives a finitely instantiating SLD-tree. Since we don't know a-priori which selection rule will have this property, it might be very expensive to check whether $A \in \mathit{FFI}$. Fortunately, as in the case of FF, it is possible to give a universal characterization of FFI in terms of fair SLD-trees. We will see, in fact, that if $A \in \mathit{FFI}$ then every fair SLD-tree for $P \cup \{\leftarrow A\}$ is finitely instantiating. To prove this result we need the following lemma which generalizes the Lemma 5.10 of [4].

Lemma 3.2.11 *Suppose there exists an infinite fair SLD-derivation for* $P \cup \{\leftarrow A_1, \ldots, A_m\}$ *with substitutions* $\theta_0, \theta_1, \ldots$. *Then for every* $k \geq 0$ *there exists* $n \geq 0$ *s.t.* $\forall i \in [1, m]$ $A_i\theta_1 \cdots \theta_n \in T_C\!\downarrow\! k$.

Proof Lemma 5.10 in [4] states a similar results, the difference is that it concludes $[A_i\theta_1 \cdots \theta_n] \in T\!\downarrow\! k$, where $[A]$ stands for the set of the ground instances of A. The proof is analogous, and proceeds by induction on k. The claim is clearly true for $k = 0$. Suppose it is true for $k-1$. Let $i \in [1, m]$. Since the derivation is fair, there exists $p \geq 0$ such that the atom $A_i\theta_0 \cdots \theta_{p-1}$

is selected in the goal G_p. Let G_{p+1} be the goal $\leftarrow B_1, \ldots, B_q$. By inductive hypothesis, for some $s \geq 0$

$$\{B_j\theta_p \cdots \theta_{p+s} \mid j \in [1,q]\} \subseteq T_C{\downarrow}(k-1)$$

holds. From the definition of T_C we have

$$A_i\theta_0 \cdots \theta_{p+s} \in T_C(\{B_j\theta_p \cdots \theta_{p+s} \mid j \in [1,q]\}).$$

Therefore, by monotonicity of T_C, we conclude

$$A_i\theta_0 \cdots \theta_{p+s} \in T_C{\downarrow}k.$$

□

Proposition 3.2.12 *Let P be a program and let $A \in \mathcal{B}_{\mathrm{V}}$. If $A \notin T_C \downarrow \omega$, then every fair SLD-tree for $P \cup \{\leftarrow A\}$ is finitely instantiating.*

Proof Suppose, by contradiction, that there is a fair SLD-tree TR for $P \cup \{\leftarrow A\}$ which is not instantiating at level k for any k. Then there exists a branch of TR which does not fail at level k for any k, and its mgu's do not instantiate $var(A)$ up to level k. This means that such a derivation is

- a refutation with computed answer substitution ε, or
- an infinite fair SLD-derivation which does not instantiate $var(A)$.

In both cases A would belong to $T_C{\downarrow}\omega$, against the hypothesis. In fact, in the former case, by strong soundness w.r.t. C-semantics (Corollary 7.1 in [39]),

$$A \in T_C{\uparrow}\omega \subseteq T_C{\downarrow}\omega$$

holds, and in the latter case, by Lemma 3.2.11 we have

$$\forall k.\ A \in T_C{\downarrow}k \ \Rightarrow\ A \in \bigcap_{k\in\omega} T_C{\downarrow}k = T_C{\downarrow}\omega.$$

□

Remark 3.2.13 *If every fair SLD-tree with $\leftarrow A$ as a root is finitely instantiating, then there exists a finitely instantiating SLD-tree for $P \cup \{\leftarrow A\}$, i.e. $A \in FFI$.*

The next theorem collects the results of Proposition 3.2.10, Proposition 3.2.12 and Remark 3.2.13 and gives the fixpoint characterization of the set *FFI*.

Theorem 3.2.14 *Consider a program P and let $A \in \mathcal{B}_{\mathrm{V}}$. Then the following are equivalent:*

a) $A \notin T_C{\downarrow}\omega$

b) $A \in FFI$

c) *Every fair SLD-tree for $P \cup \{\leftarrow A\}$ is finitely instantiating.*

Proof

$$\begin{array}{ll} (c) \Rightarrow (b) & \{\text{Remark } 3.2.13\} \\ (b) \Rightarrow (a) & \{\text{Proposition } 3.2.10\} \\ (a) \Rightarrow (c) & \{\text{Proposition } 3.2.12\}. \end{array}$$

□

Note that, analogously to (3.1), we have

$$FFI = \mathcal{B}_{\mathrm{v}} \setminus T_C\!\downarrow\!\omega.$$

By an obvious generalization of Proposition 3.2.10 and Proposition 3.2.12, it is possible to extend the previous theorem to the case of a goal consisting of several atoms.

Theorem 3.2.15 *Consider a program P and a goal $G = \leftarrow A_1, \ldots, A_n$. Then the following are equivalent:*

a) $\exists i \in [1, n]$ *s.t.* $A_i \notin T_C\!\downarrow\!\omega$,

b) *$P \cup \{G\}$ has a fair finitely instantiating SLD-tree,*

c) *Every fair SLD-tree for $P \cup \{G\}$ is finitely instantiating.*

The *FFI* set is greater than the standard *NGFF*. In fact, in [70] the *NGFF* is proved to be the complement (w.r.t. $\mathcal{B}_{\mathrm{v}}$) of the set

$$\tilde{I}_{\mathrm{v}} = \bigcap_{n \in \omega} Down(T_C\!\downarrow\!n)$$

and, by definition, $T_C\!\downarrow\!n \subseteq Down(T_C\!\downarrow\!n)$. Therefore, $T_C\!\downarrow\!\omega \subseteq \tilde{I}_{\mathrm{v}}$ holds. Hence we have:

$$NGFF = \mathcal{B}_{\mathrm{v}} \setminus \tilde{I}_{\mathrm{v}} \subseteq \mathcal{B}_{\mathrm{v}} \setminus T_C\!\downarrow\!\omega = FFI.$$

3.2.4 Model-theoretic characterization of *FFI*

We give here a validation of the NAI rule in terms of the model-theoretic semantics. First of all, we show that we need to consider a language extended by infinitely many symbols. The reason is that *Comp(P)* is not always adequate, in fact in some cases there are models of $Comp(P)$ which do not contain enough elements.

Example 3.2.16 *Consider the program*

$$\begin{aligned} P = \{\ & p(0) \leftarrow, \\ & p(s(x)) \leftarrow p(x) \quad \}. \end{aligned} \tag{3.2}$$

The atom $p(x)$ belongs to FFI (hence the NAI rule would infer $\exists\neg p(x)$) but in some models of $Comp(P)$ $\exists\neg p(x)$ is false. More precisely, it is false in those models which contain only the elements corresponding to the terms $0, s(0), s(s(0)), \ldots$.

An obvious solution is to "force" in the models additional elements. In the above example we would need just one, but in general we might need more of them.

Example 3.2.17 *Consider the program*

$$P = \{\ p(a,x) \leftarrow,\\ \quad p(x,a) \leftarrow,\\ \quad p(x,x) \leftarrow \quad \}.$$

The atom $p(x,y)$ belongs to FFI, but the formula $\exists\neg p(x,y)$ is true only in those models of Comp(P) which contain at least three distinct elements.

As the previous example shows we must consider a reference language which contains, besides all symbols which might occur in the programs, as many new constant symbols as the number of variables which might occur in the formulae we want to deal with. Since we are interested to extend the results to conjunctions of arbitrary length, it is convenient to consider a language containing infinitely many additional constants symbols $d_1, d_2, \ldots, d_n, \ldots$. We call L this language. An alternative approach would be to consider a language containing at least one constant symbol and one additional unary function symbol.

Definition 3.2.18 *The extended completion of P, $Comp_L(P)$, is defined as $IFF(P) \cup CET_L$, where $IFF(P)$ is the collection of completed definitions of predicates in P and CET_L is the set of the equality and freeness axioms ($EA \cup FA$) for L.*

$Comp_L(P)$ is an extension of $Comp(P)$, as shown by the following proposition.

Proposition 3.2.19 *Let F be a (first order) formula. Then*

$$\text{if } Comp(P) \models F \text{ then } Comp_L(P) \models F.$$

Proof L contains all symbols which occur in P, hence the axioms of $Comp(P)$ are a subset of the axioms of $Comp_L(P)$. Therefore, every model of $Comp_L(P)$ is also a model of $Comp(P)$. □

We show now that the assertion $Comp_L(P) \models \exists\neg A$ can be appended to the chain of equivalences in Theorem 3.2.14. We start with the soundness of the NAI rule.

Proposition 3.2.20 *If $A \in FFI$ then $Comp_L(P) \models \exists\neg A$.*

Proof Let $A \in FFI$ and let $\bar{x}$ be the sequence of all variables in A. Let $\bar{d}$ be a sequence of symbols not occurring in P. Then

$$\begin{array}{ll} & A \in FFI \\ \Rightarrow & \quad \{\text{Proposition 3.2.7}\} \\ & A[\bar{d}/\bar{x}] \in FF \\ \Rightarrow & \quad \{\text{Theorem 5.32 in [4]}\} \\ & Comp(P) \models \neg A[\bar{d}/\bar{x}] \\ \Rightarrow & \\ & Comp(P) \models \exists\neg A \\ \Rightarrow & \quad \{\text{Proposition 3.2.19}\} \\ & Comp_L(P) \models \exists\neg A. \end{array}$$

To prove the completeness of the NAI rule we use some results from the literature on the three-valued logic and the Kunen/Fitting immediate consequence operator Φ_P. In the following, we indicate by $\models_3$ the truth in a three-valued model. For the basic definitions of these notions see, for instance, [41] or [95] or Section 6.1.1.

Some of these results are stated into a framework which assumes an underlying language containing countably infinite sets of n-adic function symbols and of n-adic predicate symbols, for each $n \geq 0$. We need therefore to extend L to be such a language, which we call L_K. Note that, for the same reasons as in Proposition 3.2.19, for any formula F we have

$$\text{if } Comp_L(P) \models F \text{ then } Comp_{L_K}(P) \models F. \tag{3.3}$$

We use also the following relation between *FFI* and *FF*, which is the reverse of Proposition 3.2.7. Actually it is more general than the reverse, because the variables here can be replaced with ground terms whatsoever.

Lemma 3.2.21 *Let $A \in Atom$, let $\bar{x}$ be the sequence of all variables occurring in A, and let $\bar{t}$ be a sequence of ground terms. Assume $A[\bar{t}/\bar{x}] \in FF$. Then $A \in FFI$ holds.*

Proof Assume that $P \cup \{\leftarrow A\}$ has a non-failed SLD-derivation with mgu's $\theta_1, \theta_2, \ldots$ such that $\forall n. \neg Inst(\theta_1 \cdots \theta_n, \bar{x})$. Then we can replace in the goals and in the mgu's of the derivation all occurrences of variables of $\bar{x}$ by the corresponding terms of $\bar{t}$, thus obtaining a non-failed SLD-derivation for $P \cup \{\leftarrow A[\bar{t}/\bar{x}]\}$. Therefore,

$$\begin{array}{lll}
A \notin FFI & \Rightarrow & \text{every SLD-tree } TR \text{ for } P \cup \{\leftarrow A\} \\
& & \text{has a non-failed branch with mgu's } \theta_0, \theta_1, \ldots \\
& & \text{such that } \forall n. \neg Inst(\theta_0 \cdots \theta_n, \bar{x}) \\
& & \\
& \Rightarrow & \text{every SLD-tree } TR' \text{ for } P \cup \{\leftarrow A[\bar{t}/\bar{x}]\} \\
& & \text{has a non-failed branch} \\
& & \\
& \Rightarrow & A[\bar{t}/\bar{x}] \notin FF.
\end{array}$$

□

We can now prove the completeness of FFI.

Proposition 3.2.22 *If $Comp_L(P) \models \exists \neg A$ then $A \in FFI$.*

Proof Let $\bar{x}$ be the sequence of variables occurring in A.

$$\begin{array}{ll}
& Comp_L(P) \models \exists \neg A \\
\Rightarrow & \quad \{(3.3)\} \\
& Comp_{L_K}(P) \models \exists \neg A \\
\Rightarrow & \quad \{\text{Theorem 38 in [95]}\} \\
& Comp_{L_K}(P) \models_3 \exists \neg A \\
\Rightarrow & \quad \{\text{Theorem 4.1 in [67]}\} \\
& \exists \neg A \text{ is true in } \Phi_P \uparrow n, \text{ for some } n \\
\Rightarrow & \\
& \neg A[\bar{t}/\bar{x}] \text{ is true in } \Phi_P \uparrow n \text{ for some ground terms } \bar{t} \\
\Rightarrow & \\
& A[\bar{t}/\bar{x}] \text{ is false in } \Phi_P \uparrow n \\
\Leftrightarrow & \quad \{\text{Lemma 6 in [95]}\} \\
& A[\bar{t}/\bar{x}] \notin T_P \downarrow n \\
\Rightarrow & \\
& A[\bar{t}/\bar{x}] \notin T_P \downarrow \omega \\
\Rightarrow & \quad \{\text{Theorem 3.1.1}\} \\
& A[\bar{t}/\bar{x}] \in FF \\
\Rightarrow & \quad \{\text{Lemma 3.2.21}\} \\
& A \in FFI.
\end{array}$$

□

From Propositions 3.2.20 and 3.2.22 and from Theorem 3.2.14, we have the following complete characterization of the NAI rule:

Theorem 3.2.23 *Consider a program P and let $A \in \mathcal{B}_V$. Then the following are equivalent:*

a) $A \notin T_C{\downarrow}\omega$

b) $A \in FFI$

c) *Every fair SLD-tree for $P \cup \{\leftarrow A\}$ is finitely instantiating*

d) $Comp_L(P) \models \exists\neg A$.

Theorem 3.2.23 generalizes naturally to the case of a goal consisting of several atoms, i.e. A can be substituted by a conjunction $A_1, \ldots, A_n$.

3.2.5 Soundness of NAI w.r.t. Comp(P)

From Theorem 3.2.23 and Proposition 3.2.19, it follows that the NAI rule is complete also with respect to $Comp(P)$.

Corollary 3.2.24

If $Comp(P) \models \exists\neg A$ then $A \in FFI$.

Under a stronger hypothesis we can prove soundness too. Intuitively, if $P \cup \{\leftarrow A\}$ has a finitely instantiating SLD-tree and the instances on A given by this tree do not cover all possible terms of the Herbrand universe of the program, then for θ instantiating $var(A)$ to some missing terms, $P \cup \{\leftarrow A\theta\}$ should fail. Hence $\exists\neg A$ should be captured by the standard completion. The next definition formalizes this notion of "non-covering tree". In the following, if ξ is a branch of an SLD-tree, $lenght(\xi)$ stands for the (possibly infinite) number of edges of ξ.

Definition 3.2.25 *Let $A \in \mathcal{B}_V$, P be a program, TR an SLD-tree for $P \cup \{\leftarrow A\}$ and $\bar{x} = x_1, \ldots, x_n$ the variables occurring in A. We say that TR is a* non-covering tree *for A iff there exists a ground substitution σ (on the standard Herbrand universe associated to P) for A, such that for every branch ξ in TR, either*

- *ξ finitely fails, or*
- *there exists $k \le lenght(\xi)$ such that $A\sigma \notin Ground(A\theta_0 \cdots \theta_k)$, where $\theta_0, \ldots, \theta_k$ are the substitutions labeling the edges of ξ up to level k.*

Proposition 3.2.26 *If there exists a non-covering finitely instantiating SLD-tree for $P \cup \{\leftarrow A\}$, then $Comp(P) \models \exists\neg A$.*

Proof Consider a non-covering finitely instantiating SLD-tree TR for $P \cup \{\leftarrow A\}$. Let $\sigma = \{\bar{x}/\bar{t}\}$ be the ground substitution in Definition 3.2.25. By substituting verbatim $\bar{t}$ for $\bar{x}$ in TR we obtain a finitely failed SLD-tree for $P \cup \{\leftarrow A\sigma\}$. By the soundness of NAF we have $Comp(P) \models \neg A\sigma$, hence we conclude $Comp(P) \models \exists\neg A$. □

We give some examples to make clear the sense of the above proposition.

Example 3.2.27 *Consider the program in Example 3.2 and consider the goal* $\leftarrow p(x)$. *We have that the (only) SLD-tree for* $P \cup \{\leftarrow p(x)\}$ *is finitely instantiating, and covering (i.e. not non-covering). In fact, every ground substitution* θ *s.t.* $Dom(\theta) = \{x\}$ *doesn't cause failure in any branch of the tree. Therefore,* $Comp_L(P) \models \exists\neg p(x)$, *whereas* $Comp(P) \not\models \exists\neg p(x)$.

Example 3.2.28 *Consider the program P of Example 3.2.5 and consider the goal* $\leftarrow plus(x, s^2(0), y)$.

We have that the (only) SLD-tree for $P \cup \{\leftarrow plus(x, s^2(0), y)\}$ *is finitely instantiating and non-covering; in fact the edges of the (only) branch of the tree are labeled by* $\theta_0 = \{y/s(y_1)\}, \theta_1 = \{y_1/s(y_2)\}, \theta_2 = \{y_2/x\}, \ldots$ *Thus we have, for instance,*

$$plus(0, s^2(0), 0) \notin Ground(plus(x, s^2(0), y)\theta_0\theta_1\theta_2).$$

Therefore $Comp(P) \models \neg plus(0, s^2(0), 0)$, *hence* $Comp(P) \models \exists\neg plus(x, s^2(0), y)$.

3.3 Classification of the semantical characterizations

Non-ground semantics arises from the desire to provide positive programs with a declarative characterization of their operational properties (the *observables properties*). The ultimate aim of the efforts applied to positive programs should be the extension of the results obtained to general programs, i.e. logic programs possibly containing negative literals in the body of their clauses. The first step in this direction has been to characterize those atoms which "fail" from a positive program. We illustrate the evolution of the semantics of logic programs, and summarize the various results in the following table.

Ground semantics [105, 23, 7, 69]

Operational	Model-theoretic	Fixpoint operator
$SS = \{A \mid A \in B, \leftarrow A \overset{\varepsilon}{\longmapsto}{}^* \Box\}$	$P \models A, A$ ground	$A \in T_P{\uparrow}\omega = \mathit{lfp}(T_P)$
$FF = \{A \mid\ A \in B$ and there is finitely failed SLD-tree for $P \cup \{\leftarrow A\}\}$	$Comp(P) \models \neg A$	$A \in B \setminus T_P{\downarrow}\omega$

C-semantics [39, 40, 30]

Operational	Model-theoretic	Fixpoint operator
$NESS = \{A \mid \leftarrow A \overset{\varepsilon}{\longmapsto}{}^* \Box\}$	$P \models \forall A$	$A \in T_C{\uparrow}\omega = \mathit{lfp}(T_C)$
$FFI = \{A \mid$ there is a finitely instantiating SLD-tree for $P \cup \{\leftarrow A\}\}$	$Comp(P)_L \models \neg\forall A$	$A \in \mathcal{B}_{\mathrm{v}} \setminus T_C{\downarrow}\omega$

Non-ground semantics [39, 40, 70]

Operational	Model-theoretic	Fixpoint operator
$NGSS = \{A \mid \leftarrow A \overset{\vartheta}{\longmapsto} \Box\}$	$P \models \forall A\vartheta$	$A \in \bigcup_{n\geq 0} Down(T_C{\uparrow}n)$
$NGFF = \{A \mid$ there is a finitely failed SLD-tree for $P \cup \{\leftarrow A\}$	$Comp(P) \models \forall\neg A$	$A \in \mathcal{B}_{\mathrm{v}} \setminus \bigcap_{n\geq 0} Down(T_C{\downarrow}n)$

S-semantics [39, 40]

Operational	Model-theoretic	Fixpoint operator
$CSS = \{A\vartheta \mid$ A is elementary, $\leftarrow A \overset{\vartheta}{\longmapsto}{}^* \Box\}$	?	$A \in T_S{\uparrow}\omega = \mathit{lfp}(T_S)$
?	?	?

In the above table we can distinguish four different notions of success captured by different operational semantics with their own declarative counterparts. From the classical semantics to the C-semantics and up to the S-semantics, we have gradually more refined notions of success:

1. The ground success is described by the set of (ground) atoms which have an SLD-refutation (SS).

   This is the weakest notion of success, because it does not allow to distinguish between the success of an atom $p(x)$ in the program $\{p(a) \leftarrow\}$ and in the program $\{p(x) \leftarrow\}$. In fact, the least Herbrand model contains in both cases only $p(a)$, but only in the first $\{x = a\}$ is computed.

2. The operational semantics of the non-ground atomic consequences set $NESS$ defines a notion of success which, in part, solves the above problem, by including in the least Herbrand model also the non-ground atoms which have a refutation with an empty answer

substitution. Thus, in the example above, $p(x) \in NESS$ only for the second program. The success of a (possibly non-ground) atom, in this case, corresponds to the validity of the universal closure of the atom.

3. The notion of non-ground success set, *NGSS*, enlarges the precedent by allowing to capture also those atoms which have a refutation with any computed answer substitution, i.e. their existential quantification is proved. As an example, for the program

$$P = \{ \begin{array}{l} q(a,x) \leftarrow \\ p(b) \leftarrow \end{array} \} \tag{3.4}$$

   NESS contains the atom $q(a,x)$ but not the atom $q(x,x)$. This corresponds to the fact that $\forall x.q(a,x)$ is a logical consequence of P, while only $\exists x.q(x,x)$ can be deduced from it. Note that $q(x,x) \in NGSS$.

4. The real operational behavior of a logic program is fully characterized by the computed answer substitution set, *CSS*. It represents the strongest notion of success, by capturing the difference between answers effectively computed and answers obtained by instantiation of universally quantified variables, thus exactly characterizing the notion of computed answer substitution.

   For the program (3.4), $q(a,b)$ does not belong to *CSS*. This corresponds to the fact that $\{x = a, y = b\}$ is not computed by starting the derivation with $\leftarrow q(x,y)$. Note that $q(a,b)$ belongs to *NESS*.

This initial theory has been successively extended to some other observable properties, namely the non-ground finite failure set (*NGFF*, [70]) and the failure by finite instantiation set (*FFI*, [32, 30] or Section 3.2.1) which, are the operational concepts corresponding to *NGSS* and *NESS*. Their fixpoint characterizations reflect the standard one of the *FF* as the complement of $T_P \downarrow \omega$, and both exploit the T_C operator.

In order to complete the picture in this table we need a concept of non-ground failure able to represent the dual of the computed answer substitution set *CSS*. To this aim, we would like to exploit the complement of $T_s \downarrow \omega$ with respect to the Herbrand base with variable $\mathcal{B}_{\mathrm{v}}$. However, by investigating the operational meaning of this set we realized that this correspondence is not so immediate. On the contrary, it is very difficult to declaratively define a set satisfying the desired property, even if extensions of T_s are considered. Intuitively, the dual notion of the computed answer substitution (modeled by the *CSS* set) should be that of "non-computed answer substitution", i.e. a substitution which cannot be obtained as an answer in a finite or infinite derivation starting from an elementary atom. Therefore, it could be useful to dispose of a model for infinite computations. This can be achieved by means of a *symmetrical* theory.

3.3.1 Towards a symmetrical theory

The existing fixpoint characterizations of the various failure sets defined for a given program P is obtained by a downward iteration of the appropriate transformation T associated to P (T_P, T_C) until ω^3. Dually, the successes are described by the upward iteration of T until ω. However, while the latter corresponds to the least fixpoint of T, $lfp(T)$, the former is not, in general, the greatest fixpoint of T, $gfp(T)$. This asymmetry has been pointed out

[3] Even if it is not immediately evident, at most ω iterations have to be performed also for *NGFF*, since a countable intersection has to be considered in this case.

in [58], and in [53] programs satisfying $T \downarrow \omega = gfp(T)$ are called *canonical*. By recalling that the Herbrand models of $Comp(P)$ are the fixpoints of T, this asymmetry means that we can restrict to Herbrand models only for testing if an atom A is a logical consequence of $Comp(P)$, but we have to consider all models for testing if $\neg A$ is a logical consequence of $Comp(P)$ (i.e. A does not belong to any model of $Comp(P)$), since $A \notin gfp(T)$ does not imply $A \notin T \downarrow \omega$, (in general $gfp(T) \subset T \downarrow \omega$ holds). The pleasant aspect of this fact is that effectivity is always achieved: the complement of $T \downarrow \omega$ is recursively enumerable whereas in general the complement of $gfp(T)$ is not. The unpleasant aspect is that Herbrand models are not the representatives of all models of $Comp(P)$. On the other hand, if the fixpoint operator associated to the program satisfies the property $T \downarrow \omega = gfp(T)$ (i.e. T is not only continuous, but even downward continuous), then both computability and representativeness is obtained. Therefore, eliminating the asymmetry in the fixpoint semantics has been the purpose of many researchers persuaded of the fact that any "decent" program should enjoy the above mentioned property. In a sense, all definite programs are canonical. In fact, in [53] it is shown that every definite program can be appropriately extended to be canonical, and [12] shows that if the Herbrand universe is replaced by a suitable algebra, then we can get rid of the asymmetry. [70] follows a similar approach to define a greatest fixpoint characterization of the non-ground finite failure set, *NGFF*. The basic idea is very interesting and it suggests some useful intuitions. In order to get a symmetrical characterization of the successes and the failures of a program P, downward continuity of the fixpoint operators associated to P is required. This is obtained if the domain contains some infinite elements corresponding to those elements (the limits) needed for "completing" the underlying structure to be a topologically closed structure. But then the semantics becomes potentially able to capture also some infinite behaviors of the program. In other words, any fixpoint semantics which is based on a symmetrical theory should offer the possibility to model some infinite computations of programs. On the other hand, to capture divergence it is in general necessary to adopt a more refined semantical domain. This possibility is not taken into account in the first three semantics (ground, non-ground and C-semantics) which consider only terminating computations (finite failures or finitely instantiating computations). For them, a non-symmetrical characterization of successes and failures is then sufficient. Starting from the intuition that S-semantics is the most "truthful" description of the behaviors of programs, we think that any characterization of them using S-semantics cannot disregard to capture also some non-terminating computations of programs. Our contention is that, in order to complete the table, we should tend towards a symmetrical theory.

Chapter 4

Infinite computations

In which we see a general language which subsumes logic programming and constraint logic programming languages. A notion of observables for this language includes success, failure and infinite behavior.

The matter of symmetry discussed at the end of Chapter 3 naturally introduces the problem of infinite elements, for if they are present in the domain of the fixpoint operator, then its greatest fixpoint can be reached at most in ω steps. There are various possibilities of considering something to be infinite. Usually, infinite objects are thought as mathematical objects which are computed through the techniques of the so-called "lazy evaluation". A typical and well-known example is that of a stream

$$(a_0, (a_1, (a_2, \ldots))),$$

which at any finite stage of its development looks like an initial segment of an ordinary list

$$(a_0, (a_1, (\ldots (a_{n-1}, \mathit{nil}) \ldots))),$$

but differs by proceeding indefinitely.

An even simpler example is the infinite natural number $s(s(s(\ldots)))$, the successor of the successor of the successor, etc. in infinitum.

Another traditional way to think of infinity is that of sets which contain infinitely many elements, possibly infinite in their turns.

However, infinite elements of sets and infinite sets in the sense described above are not the only possible expressions of the notion of infinity. Mathematics offers a lot of examples of objects which are infinite but not in the usual sense. The iterative set

$$\{\{\{\cdots\}\}\},$$

is such an example. It is the singleton set whose only element is the singleton set whose only element is the singleton set, etc. in infinitum and represents the simplest form of non-well-founded set in the sense of Aczel ([2]).

The idea of infinity is implicit also in sets of the form

$$A \times (A \times (A \times \ldots)),$$

where A may itself proceed indefinitely, or even

$$((\ldots) + (\ldots)) + ((\ldots) + (\ldots)),$$

the disjoint union of two sets, each of which is the disjoint union of two sets, each of which is the disjoint union of two sets, etc. in infinitum. The reason why these sets are infinite is not that they contain infinitely many elements, but it depends on the fact that they have an infinite depth or proceed indefinitely.

The two (relatively) recent theories that have been contrived precisely for dealing with these various kinds of infinite objects are the Scott theory of domains for denotational semantics and the Aczel theory of non-well-founded sets. Of these, domain theory indeed succeeds in making sense of streams and similar infinite objects, but it does not harmonize with the interpretation of the set-theoretical laws in the intuitionistic type theory such as that of Martin Löf ([74]). The reason is mainly due to the definition of the fixpoint operator which is a central notion in domain theory. Roughly speaking, when the formal laws defining the fixpoint operator are added to the type theory, this becomes inconsistent, as they have the same effect as introducing an infinite natural number subject to the axioms

$$\infty \in \mathcal{N},$$

$$\infty = s(\infty),$$

where s is the successor function. Since it can be shown that the last circular definition is contradictory, it is inadmissible in type theory. Aczel's approach avoids this contradiction by introducing non-well-founded sets which obey some axioms that are opposite to the axioms of standard set-theory. Analogous motivations inspired other approaches to the semantics of infinite objects, all based on the non-standard arithmetic and analysis. The interested reader can find useful references and explanations on this subject in [75].

Here we concentrate on the aspects related to the semantics of infinite objects regarded as "objects computed at infinity" (the streams above mentioned), and we are interested in studying the models for logic programs which are able to take them into account. In doing so, the Scott theory of domains will come very in handily, its purpose being just that of establishing a notion of computability with respect to domains containing infinite elements.

Limits of computations

If we reflect about the kind of objects mentioned above, we can recognize their common nature with concepts already studied since a long time by mathematician. Old examples like infinite decimal fractions or infinite continued fraction expansions proceed indefinitely in just the same way as streams. The mathematics that has been developed in order to deal rigorously with these particular infinite objects, namely calculus and analysis, have gradually evolved towards the notion of *limit*. In a computational scheme where some algorithms are employed successively to gain increasingly refined approximations to the desired result, no more suitable notions seem better to represent the "final result". This will be the "limit" of the sequence of the approximations. Of course, it is needed to give a precise definition of how well some "stage" of the computation approximates the limit. The more immediate approach to this question is the metric one: the notion of distance tells us immediately how close an approximation is to the desired limit. An alternative procedure is to think of an approximation as a subset in which the result lies. The smaller the set, the better the approximation, in the sense that the smaller set gives "more information". This approach leads naturally to the use of an order-theoretic language: one defines, in place of a metric, a notion meaning roughly that an element y is a "finite approximation" to an element x, namely y is less than or equal to x as elements of a partially ordered set. In both cases, we have a notion of limit of a computation which can be used to define the semantics of a given program even when it is recursively defined (and then it could cause non-termination).

Actually, metric spaces and cpo's represent the main alternative approaches to define "domains" for denotational semantics, that is those particular abstract structures whose elements are used to ascribe meanings to syntactical objects (the programs), following the syntactical structure of these objects.

The notion of limit in these domains is represented by means of the fixed points of a second-order functional. Both for cpo's and for metric spaces we have a convenient fixed point theorem with the aid of which it is possible to interpret recursive definitions. The applications of these theorems are so overspread and important in this approach to the denotational semantics of programming languages that it is sometimes called the fixed-point approach.

A "concurrent" view of infinite computations

Difficulties in modeling infinite computations come out, above all, in the framework of concurrent systems, due to the presence of non-determinism. When a program is "deterministic", i.e. no non-deterministic choice constructs are allowed to occur in it, so that the course of a computation is completely specified at any stage of the computation, its behavior can be characterized neatly as the minimal fixpoint of an appropriate continuous function ([61]). Therefore, in the deterministic case it is possible to obtain a denotational semantics by means a simple application of elementary domain theory. As an example, for the deterministic fragment of concurrent constraint programming (ccp) an elegant denotational semantics based on closure operators has been defined in [89]. However, if non-determinism is added as a possible constructor in the programming language (thereby obtaining a concurrent indeterministic programming language), then the semantics of programs is not always obtainable by naïvely extending the semantics for deterministic programs to cope with the new operator. In particular, this is not possible when the kind of non-determinism is *don't care*, i.e. a concurrent process is required to select one of the alternatives which are enabled at a certain moment, and to "commit" to that choice, without any possibility of backtracking. The main source of troubles is the need of assuming *fairness* as a condition that any execution of a non-deterministic (and/or parallel) program must satisfy. Fairness is an infinitary property, in the sense that it cannot be detected at finite time, and in its simplest form it states that no component of a system is delayed indefinitely for execution. It is important noting that in case of don't care non-determinism this condition involves *all* components of the system, and then also those components occurring in an alternative choice context. For the latter the possibility to be selected sooner or later could not otherwise be taken for granted because of the absence of backtracking. The problem of fairness is closely tied to infinite computations, since the correctness of a program w.r.t. almost all properties (even the finitary ones) cannot be proved unless the infinite computations are restricted to just the fair ones. Thus, the issue of fairness cannot be easily ignored. On the other hand, the possibility of infinite computations arises naturally and is often desired just in concurrent systems such as operating systems or dataflow networks (the so-called reactive systems) where non-determinism is an essential as well as natural aspect. Consequently, an adequate treatment of infinite computations should not ignore fairness. This condition appears to present a number of fundamental problems mainly due to its inherent connection with the notion of "unbounded non-determinism", i.e. the non-determinism given by choice constructs with infinitely many alternatives. More precisely, as shown in [6], under the assumption of fairness it is always possible to translate every non-deterministic program (and then also those with finitely many alternatives) into an appropriate unbounded non-deterministic one. The unpleasant point is that unbounded non-determinism is known to cause the lack of continuity of the various semantic functions, and in [6] it is formally shown that no reasonable continuous semantics can exist for it. Thus, the rich variety of models due to Scott, Plotkin et al. is not apparently available to give an appropriate semantics for concurrent programs, and resorting to the use of monotonic non-continuous functions ([79]), or to category-theoretic methods ([78]) seems to be the best alternative. Other recently proposed approaches reserve to infinite computations a treatment which is often unsatisfactory due to the above mentioned difficulties in dealing with non-determinism in presence of fairness, as it does not capture the real infinite essence of those computations, in some sense. In two recent works on the semantics of ccp, infinite computations are treated either under the slogan "divergence is catastrophic", thus confusing processes which can fail with processes which can engage infinite executions (this is the case of the semantics for indeterminate ccp of [89]), or they are introduced for "convenience" in order to correctly model finite computations (this is the case of [26]).

Different considerations have to be made for concurrent languages where the kind of non-determinism is *don't know*, i.e. it is meant for an exhaustive search algorithm which considers all the possible (in the sense of enabled) alternatives. This is the case of the angelic ccp language defined in [59], which subsumes both logic programming and constraint logic programming. Here, the assumption of fairness is needed only for the parallel construct (and-fairness), but it is not meaningful for the non-deterministic choice construct. This fact radically changes the situation because the above negative results do not apply any more. Therefore, the techniques based on cpo's and continuous functions (powerdomain constructions, metric completions, etc.) are still practicable for defining a denotational semantics for logic programming (as a particular case of angelic ccp).

In this chapter we focus on the problem of characterizing infinite *and-fair*, computations of logic programs via a fixpoint (denotational, bottom-up, declarative, ...) semantics. After reviewing the main approaches occurring in the literature, we introduce a language generalizing both logic programming and constraint logic programming, which turns out to be more convenient for our treatment. Moreover, in developing our semantics we will exploit the techniques typically used to define the denotational semantics of programming languages (see for instance the Stoy's book [101]). Reasoning in terms of processes instead of atoms, that is assuming a process interpretation of (constraint) logic programming, will result very helpful in modeling infinite computations which (after all) represent a notion closer to traditional programming languages than to pure logic programming language.

4.1 The "infinite" in logic programming

From a purely theoretic point of view, we can think that the problem of infinite computations was born in logic programming from the desire of making a sense to all the atoms belonging to the Herbrand base. This can be considered as a big onion whose strata hide each one a particular set of atoms with a certain operational characterization. The core of the onion is the least Herbrand model which contain all successful atoms. Van Emden and Kowalski showed that it could be "constructed" by means of the transformation T_P associated to the program P, by proving that the minimal Herbrand model coincides with the least fixpoint, $T_P \uparrow \omega$, of T_P. By starting from the opposite end, Lassez and Maher continued the exploration of the Herbrand base and discovered that the complement (w.r.t. the base) of $T_P \downarrow \omega$ contained all the finitely failed atoms. But, what did $T_P \downarrow \omega \setminus T_P \uparrow \omega$ contain? Since the previous results concerned the finitary case, it was obvious to deduce that the question could be answered by trying to characterize the infinite computations.

The main approaches to this problem which have been developed in the literature can be grouped essentially in three different kinds: metric, order-theoretic and constraint-based. Of these, the metric and order-theoretic approaches have in common a property of "canonical-hood", in the sense that they both try to obtain the identification $gfp(T_P) = T_P \downarrow \omega$, the property of those programs that Jaffar and Stuckey call *canonical*.

The constraint-based approach is, in a sense, more declarative; it exploits the partial nature of constraints instead of resorting to metric limits or lub's of chains of sets.

Of course, a natural requirement for modeling infinite computations is the presence of infinite elements in the universe of the discourse. As we already mentioned, this very requirement needs for obtaining downward continuity of the operator T_P, the condition under which the greatest fixed point $gfp(T_P)$ can be reached in ω steps. In fact, for example, for the program

$$\begin{aligned} P = \ & \{p(a) \leftarrow q(x), \\ & q(f(x)) \leftarrow q(x)\}, \end{aligned} \tag{4.1}$$

$gfp(T_P) = \emptyset$, whereas $T_P \downarrow \omega = \{p(a)\}$, and it is easy to see that, if the base contains an element of the form $q(f^\omega)$ with the property $q(f^\omega) = q(f(f^\omega))$, then we can obtain $gfp(T_P) = T_P \downarrow \omega = \{p(a), q(f^\omega)\}$.

Beyond the considerations concerning symmetry that have been pointed out in Section 3.3.1, we can observe that both the atoms $p(a)$ and $q(f^\omega)$ are obtained by means of infinite fair computations. This gives an intuition of the idea which has led to the first two kinds of approach, whose essence can be sketched as follows. First turn the domain into a "complete domain" by adding the infinite elements; then define appropriately the transformation T_P so that the desired property is satisfied.

4.1.1 The metric approach

The metric approach is due to N. Abdallah ([77]) and J. Lloyd ([71]) and it is based on the "infinite-term" semantics introduced in [3] and the notion of metric completion of the Herbrand universe introduced in [8].

In [77] the Herbrand Universe is made a metric space by introducing a distance between terms as in [8]. The infinite terms are obtained by completing the metric space, i.e. by adding all the limits of Cauchy sequences of terms. In [71] infinite terms are defined to be infinite trees as in [3], and a similar distance as in [77] is defined on the complete Herbarand Universe, i.e. the Herbarand Universe extended with infinite terms. In both cases the result is an Herbrand base containing also infinite atoms (the complete Herbrand base) which is shown to be a compact metric space (w.r.t. the obvious extension to atoms of the distance between terms). By exploiting the metric properties of the Herbrand base, the transformation T_P extended on the complete interpretations (subsets of the complete Herbrand base), T'_P, is shown to satisfy $T'_P \downarrow \omega = gfp(T'_P)$.

However, concerning the relationship between infinite computations and the greatest fixpoint of the extended operator, it is not quite satisfactory. In particular, in [77] the soundness and completeness of infinite fair derivations is given in terms of a generalization of the notions of correct answer substitution and computed answer substitution of [7]. In [71] infinite computations are called *perpetual processes* and their intended meaning is that of (in Lloyd's words) "computations which do not terminate and yet are performing useful work, in some sense". The sense of an infinite computation being useful is given by the notion of atom *computed at* *infinity*. However, this notion is not completely captured by $gfp(T'_P)$: there are atoms which are not intuitively computable at infinity and yet they belong to $gfp(T'_P)$. One problem is that the construction of the greatest fixpoint does not reflect how the infinite terms are constructed during the computation. Thus an infinite atom obtained in this construction may be not obtainable operationally. For example, with the program $P : \{p(x) \leftarrow p(x)\}$, the infinite atom $p(f^\omega)$ is not "computed at infinity". It is anyway captured by the greatest fixpoint construction (as well as $p(t)$ for all ground terms t). Another problem is more tied to the metric nature of the approach, as it derives from the fact that the notion of (metric) limit does not allow to always distinguish between two sequences (derivations) which are operationally distinct. For example, the sequences

$$\{p(f^n(x), f^n(x))\} \qquad \text{and} \tag{4.2}$$

$$\{p(f^n(x), f^{n+1}(x))\} \tag{4.3}$$

converge to the same metric limit $p(f^\omega, f^\omega)$, despite the fact that only the latter reflects the derivation sequence for the goal $\leftarrow q(x)$ with the program

$$\begin{aligned} P_1 = \{\ & p(f(x), f(f(x))) \leftarrow p(x, f(x)) \\ & q(x) \leftarrow p(x, f(x))\}. \end{aligned} \tag{4.4}$$

The consequence is that the program P_1 is not distinguished from the program

$$\begin{aligned} P_2 = \{\ & p(f(x), f(x)) \leftarrow p(x, x) \\ & q(x) \leftarrow p(x, f(x))\}. \end{aligned} \tag{4.5}$$

even though the goal $q(x)$ fails with P_2 due to the occur check. This problem can be solved by considering complete partial ordered sets (cpo's) with the associated notion of least upper bound (lub), instead of metric spaces and limits. In fact, it is well-known that in general the former have more "discriminating power" than the latter, since they identify less elements.

4.1.2 The order-theoretic approach

The second kind of approach is due to Golson and is based on a different technique of completion, namely the completion by ideals. In general, given a partial ordered set D, an ideal I is a subset of D which is directed (every pair of elements of I has a lub in I), and downward closed (if x is an element of D which is less than or equal to some element of I, then x belongs to I). It is well-known that the sets of all the ideals of D ordered by set-inclusion is a complete partial order. This ensures that every chain of ideals has a lub which is again an ideal, representing its limit.

In [49] the ideals of the Herbrand base are represented as sets of the form $A\Theta$, where A is a (finite) atom, Θ is a directed set of substitutions and $A\Theta$ is the sets $\{B \mid \exists\sigma \in \Theta . B \leq A\sigma\}$. These are called *objects*. An infinite object is just an object $A\Theta$ where the cardinality of Θ (modulo renaming) is infinite. The representation of the ideals by objects contains information concerning the "unifiablity". For example the sequences 4.2 and 4.3 are now sequences of objects respectively of the form

$$A_1\Theta_1 = p(y, y)\{\{y/f^i(x)\}\}_i,$$

and

$$A_2\Theta_2 = p(y, f(y))\{\{y/f^i(x)\}\}_i.$$

The difference w.r.t. the unification is represented by the "fixed" components of the objects, namely A_1 and A_2. This allows to distinguish the two sequences. In fact, since $p(y, y) \in A_1\Theta_1$, but $p(y, y) \notin A_2\Theta_2$ (and symmetrically, $p(y, f(y)) \in A_2\Theta_2$, but $p(y, f(y)) \notin A_1\Theta_1$), they have different lub's. Hence the programs 4.4 and 4.5 of the previous section are distinct in this approach.

Once the completion of the Herbrand base has been carried out, the Golson's method proceeds analogously to the metric method. A fixpoint transformation, S_P is now defined on the complete lattice of interpretations, i.e. $\subseteq$-upward closed sets of objects, with the property $S_P{\downarrow}\omega = gfp(S_P)$. However, this construction works only for programs whose clauses contain at least one global variable, since this condition is necessary for the continuity of S_P. Hence programs like 4.1 are excluded in this approach. Moreover, infinite computations are not yet completely characterized, as the greatest fixpoint captures only the *minimal infinite objects*, i.e. the most general objects constructible by non-terminating computations.

4.1.3 The constraint-based approach

The identification of $T_P{\downarrow}\omega$ with $gfp(T_P)$ seems reasonable since, as shown in [53], every Horn definite program is "equivalent" in some sense to a canonical program, or, as asserted in [12], "definite clause programs are canonical over a suitable domain".

However, in general the partition of the Herbrand base is more complex and another set has to be considered, namely the difference set $T_P{\downarrow}\omega \setminus gfp(T_P)$. From the characterizations of the models of $Comp(P)$ and of the finite failure set, we know that this stratum is constituted by those atoms which are false in every Herbrand model of $Comp(P)$, (since they are within the complement of the maximal Herbrand model of $Comp(P)$), and yet they are true in at least one (non-Herbrand) model of $Comp(P)$, (since they are not finitely failed). It is named the Herbrand gap and in [57] it has been provided with another characterization based on the notion of *ground finite failure*. Under this notion a ground atom A is said to finitely fail if all branches of its *ground*-derivation tree end in failure after a finite number of steps, where a ground-derivation is simply an SLD-derivation from the program P_g obtained by replacing every clause in P with its ground instances. What distinguish ground finite failure from ordinary finite failure, and makes it non-computable, is that there may be infinitely many clauses of P_g with a given head, so that the derivation tree can have infinitely many branches from each node. Consequently, the König's lemma fails, and although each branch of the tree is finite, the whole tree may be infinite. However, this characterization hides the "infinite" essence of the atoms of the Herbrand gap, as they are not regarded as goals having an infinite computation. This appears more evident by another characterization of that set, which has been given in [55] within the framework of constraint logic programming. Here, thanks to the separation of the atom (representing the goal) from its solutions (expressed by constraints), it is possible to distinguish between two kinds of infinite computations corresponding respectively to what Jaffar and Lassez call the *infinite failure* and the *infinite success*. Recalling that in clp a derivation step produces a constraint (instead of a substitution), the result of an infinite fair derivation from a unitary goal (atom) is just the union of the constraints stored during computation (the limit answer constraint). If this has a solution in the structure interpreting the domain of constraints, then the infinite derivation is said to succeed, otherwise it is said to fail. The result of this distinction is an even more complex partition of the Herbrand base B, which is now regarded as composed by four strata:

- the success set, corresponding to $lfp(T_P)$;

- the infinite success set corresponding to $gfp(T_P) \setminus lfp(T_P)$;
- the infinite failure set corresponding to $T_P{\downarrow}\omega \setminus gfp(T_P)$;
- the finite failure set corresponding to $B \setminus T_P{\downarrow}\omega$.

4.2 A general framework

If we are only interested in the (finite) successes and the finite failures a logic program P, then observing the mere input-output behavior of P is all what we need. But if we are concerned with infinite computations, a less abstract kind of observation has to be chosen to provide a satisfactory account of this phenomena. In particular, we have to observe the partial results produced at any stage of the computation, so to define the results of infinite computations as limits. The ease of defining a model-theoretic semantics for success and finite failure of a logic program is in part due to the fact that we have just to model some "external" behaviors of the program: how a computation is performed does not matter in determining its success or finite failure. Symmetrical reasons explain why a model-theoretic characterization of infinite computations should be more complicated. On the other hand, for the observation step-by-step of a computation it is very handy regarding a logic program in its purely programming guise, i.e. adopting a "process view" of logic programs and goals. In this sense, a denotational-like approach seems to be the more natural one. These considerations suggested us to try a treatment of infinite computations by relying on the so-called *process interpretation* ([104, 92]) of logic programming language. In this interpretation the goal is a system of parallel processes (the atoms), the shared variables correspond to the communication channels, different clauses defining the same predicate provide non-determinism, the resolution step corresponds to a transition of the system, and the unification represents the mechanism by which processes communicate: variables occurring in two or more atoms correspond to communication channels between the associated processes, and their instantiation, due to unification, corresponds to a communication action. Thus, logic programming becomes a special case of the more general paradigm of concurrent logic programming.

A further generalization turns out very useful to our aim, namely that of reasoning in terms of constraints over an arbitrary domain, instead of substitutions, i.e. equalities over the Herbrand Universe. In fact, the ability to represent and manipulate partial information about the domain of discourse (the domain where computations are performed) allows us to easily define the result of an infinite computation as the limit of its finite approximations (partial results). Thus, we will assume the general paradigm of concurrent constraint programming, and, in this framework, we will define a language (the Γ_clp language) with an expressive power in between ccp and clp. Before describing this language we have to introduce the notion of *constraint system*, which plays a central role in ccp (and then also in our language). Both the language and the semantic construction ([33]) we will develop are parametric with respect to it.

4.2.1 Constraint Systems

Constraint systems represent the domains where computations have to be performed. Since our attention is turned over infinite computations, it is convenient to dispose within the domain of some infinite elements. These can be obtained as the maximal elements in a system of partial information where the (partial) ordering is used to express the fact that an element x is less defined, i.e. more partial, than another element y. This is indeed the notion of *information* *system* of Scott ([90]) and has been used in [88] for defining the constraint systems underlying ccp languages.

Roughly, an information system is based on a set of "propositions" with an entailment relation subject to a set of axioms. The *elements* of an information system are sets of propositions which are consistent and closed under entailment.

In [88] a constraint system is defined as an information system, but for the requirement of consistency, which is removed. This is necessary in ccp in order to capture the possibility of a program giving rise to an inconsistent state during its execution. In our case this would not be necessary. However, we maintain this extension because this approach leads to constraints systems which are complete algebraic lattices. Constraints systems having this property are very desirable domains since their powerdomains can be algebraically characterized in terms of some family of sets instead of a family of sets of sets ([81]). In Section 4.4 we will use this property to define a simple denotational semantics for the language Γ_clp.

Moreover, in [88] some operators to express hiding of information are introduced. From a programming point of view, these operations allow the introduction of parameter passing in constraint programming languages, and the introduction of "local variables" within procedures.

We regard a constraint system as a complete algebraic lattice in which the ordering $\leq$ is the reverse of the the entailment relation $\vdash$, the top element *false* represents the inconsistent constraint, the bottom element *true* the empty constraint, and the lub operation $\sqcup$ the join of constraint, or logical and. We refer to [88] for more details about the construction of such a structure.

Definition 4.2.1 *A constraint system is a complete algebraic lattice* $(\mathcal{C}, \sqsubseteq, \sqcup, \mathit{false}, \mathit{true})$ *where* $\sqcup$ *is the lub operation, and true, false are the least and greatest elements of* $\mathcal{C}$*, respectively.*

Following the standard approach, we will sometimes use $\vdash$ instead of $\sqsubseteq$. Formally $c \vdash d \Leftrightarrow d \sqsubseteq c$. By $K(\mathcal{C})$ we will indicate the algebraic (or finite) elements of $\mathcal{C}$.

In order to model information hiding a sort of existential quantifier for goals is needed. This can be formalized by introducing a kind of constraint system which supports *hiding operators*, a notion borrowed from the theory of cylindric algebras, due to Henkin, Monk and Tarski ([51]).

Assume given a (denumerable) set of variables *Var* with typical elements $x, y, z, \ldots$, and consider a family $\{\exists'_x \mid x \in \mathit{Var}\}$ of operators $\exists'_x : K(\mathcal{C}) \to K(\mathcal{C})$, which map a finite constraint c into its existential closure $\exists'_x c$, and obey the following axioms:

(i) $\exists'_x c \sqsubseteq c$,

(ii) if $c \sqsubseteq d$ then $\exists'_x c \sqsubseteq \exists'_x d$,

(iii) $\exists'_x (c \sqcup \exists'_x d) = \exists'_x c \sqcup \exists'_x d$,

(iv) $\exists'_x \exists'_y c = \exists'_y \exists'_x c$.

Since $\exists'_x$ is monotone on compact elements, its continuous extension to $\mathcal{C}$ exists and is unique. Thus, we can define $\exists_x : \mathcal{C} \to \mathcal{C}$ to be the continuous extension of $\exists'_x$ to $\mathcal{C}$.

From (i) and (iii) it follows that

(v) $\exists_x \exists_x c = \exists_x c$.

This means that $\exists_x$ is a *kernel* operator, i.e. a function which is *decreasing* (i), *monotonic* (ii) and *idempotent* (v).

Starting from a constraint system $\mathcal{C}$, construct a cylindric constraint system $\mathcal{C}'$ by taking $\mathcal{C}' = \mathcal{C} \cup \{\exists_x c \mid x \in Var,\ c \in \mathcal{C}'\}$ modulo the identities and with the additional relations derived by the above given axioms.

In order to model parameter passing, it will be useful to enrich the constraint system with the so-called *diagonal constraints*, also from Henkin, Monk and Tarski ([51]).

Given a cylindric constraint system $\mathcal{C}'$, define a diagonal constraint system $\mathcal{C}''$ by taking $\mathcal{C}'' = \mathcal{C}' \cup \{d_{xy} \mid x, y \in Var\}$ modulo the identities and with the additional relations derived by the following axioms:

1. $d_{xx} = true$,
2. if $z \neq x, y$ then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
3. if $x \neq y$ then $c \sqsubseteq d_{xy} \sqcup \exists_x(c \sqcup d_{xy})$.

Intuitively, a constraint d_{xy} expresses the equality between x and y. Used together with the existential quantification these constraints allows us to model the variable renaming of a formula ϕ. In fact, thanks to the above axioms, the formula $\exists_x(d_{xy} \sqcup \phi)$ has precisely the meaning of ϕ with al the free occurrences of x replaced by y, i.e. $\phi[y/x]$.

In the following $\mathcal{C}$ will indicate any diagonal (and then cylindric) constraint system.

4.2.2 The language Γ_clp

The choice of the particular language we present in this section is aimed at providing an appropriate support to the description of the model which we propose to extend the standard semantics of logic programs (i.e. success and failure semantics) by capturing the notion of "results" of infinite computations.

The first step is moving to the clp paradigm, by allowing a generic constraint system (as defined in Section 4.2.1) to be embedded in the language. The use of generic constraints instead of substitutions shall remove some difficulties already come out in previous approaches to the semantics of infinite computations in pure logic programming. Here (partial and limit) answers are expressed by means of (compositions of) most general unifiers which represent solutions to equalities constraints over the Herbrand universe. Usually, this is extended to contain some infinite elements, (e.g. ω), having some limit properties (e.g. $\omega = f(\omega)$), and it may happen that an infinite element is solution of a constraint whereas its finite approximations are not (e.g. $f^n(x)$ is not a solution of $x = f(x)$ due to the occur check, whereas f^ω is a solution of that constraint for it is a limit element). This causes incompleteness since limit elements which are not constructible operationally may be considered in the semantic model. On the other hand, in clp derivations are defined by relying on the notion of consistency check which abstracts from the particular solution by which a constraint is satisfied (it is consistent), so that one can reason in terms of the constraints themselves rather than of their solutions.

We briefly recall the syntax and the computational mechanism of clp. Given a constraint system, a clp goal is a construct of the form

$$:-\ c_1, \ldots c_m, B_1, \ldots, B_m$$

where the c_i's are finite constraints such that $c_1 \sqcup \ldots \sqcup c_m$ is consistent, and A and the B_i's are *atoms*, i.e. predicates applied to terms. Usually the constraint system fixes the interpretation of the terms, but not of the predicates. The predicates are defined by a program, which consists of a set of *clauses* of the form

$$A\ :-\ d_1, \ldots d_n, A_1, \ldots, A_n$$

where the d_i's are finite constraints such that $d_1 \sqcup \ldots \sqcup d_n$ is consistent, and A, A_i's are atoms. The set of clauses whose left-hand sides have predicate p constitute the definition of p. Starting from a goal, a computation evolves by replacing each time an atom having predicate, say, p, by the body of a clause defining p, nondeterministically selected among those which give a consistent resulting constraint. A clause is renamed each time it is used, so to avoid variable clashes with the goals. More formally, given a renamed clause as the above one, a *computation step* rewrites a goal

$$:- \ c_1, \ldots c_m, B_1, \ldots, B_m$$

into

$$:- \ c_1, \ldots c_m, d_1, \ldots d_n, B_1, \ldots, B_{i-1}, A_1, \ldots, A_n, B_{i+1}, \ldots, B_m$$

provided that the constraint $(B_i = A) \sqcup c_1 \sqcup \ldots c_n \sqcup d_1 \sqcup \ldots \sqcup d_n$ is consistent[1], i.e. different from *false*, (*consistency check*).

The result of a finite computation ending in a goal of the form $: - \ c_1, \ldots, c_n$ (successful computation) is $c_1 \sqcup \ldots \sqcup c_n$. The result of an infinite (and-fair) computation is the least upper bound of the constraints generated at some step. The results of a goal are the results of all its computations.

These definitions are very simple, but they do not provide a suitable basis to our model. By following a usual approach to the semantics of programming languages, we consider a structured operational semantics, based on a transition system, by means of which we define the notion of observables as the set of final constraints resulting from all possible derivations. Then we develop a compositional semantics, based on logical operators, and show its full correspondence with the observables. It turns out that it is more convenient to reason about a generalized language structured as the free language generated by a BNF grammar. We reformulate the syntax of clauses and goals by means of the grammar described in Table 4.1. The resulting language will be called Γ_clp. In the following we will assume that there is at most one declaration for each predicate.

Constraints and atoms are basic constructs also in our language, but we restrict to atoms of the form $p(x)$. The goal c represents a basic (finite) constraint in some constraint system $\mathcal{C}$. This is a parameter of the language and then also of the semantic construction we develop in the following. Furthermore we need to represent the conjunction of atoms, the alternative choice among clauses, and locality. Correspondingly we introduce the operators $\wedge$, $\vee$ and $\exists_x$. The choice of these symbols is because we have in mind a denotational semantics which assigns to goals a logical meaning, and the idea is that $\wedge$, $\vee$ and $\exists_x$ will correspond to the and, the or and the existential operator respectively. The $\exists_x$ symbol here must not be confused with the analogous operator of the constraint system, but, of course, there is a close correspondence among them.

Note that Γ_clp subsumes clp. For instance a declaration for p consisting of two clauses

$$\begin{array}{lll} p(x,a) & :- & c(x), q(x,y) \\ p(b,x) & :- & \end{array}$$

can be rewritten in Γ_clp as

$$\begin{array}{lll} p(z) & :- & \exists_x(z = \langle x, a\rangle \wedge c(x) \wedge \exists_y \exists_w (w = \langle x, y\rangle \wedge q(w))) \\ & & \vee \\ & & \exists_x(z = \langle b, x\rangle). \end{array}$$

[1]The underlying constraint system is assumed to contain equality, and $p(t_1, \ldots, t_m) = q(u_1, \ldots, u_n)$ stands for $t_1 = u_1 \sqcup \ldots \sqcup t_m = u_n$ if $p = q$ and $m = n$, $false$ otherwise.

Programs	$P ::= \epsilon \mid D.P$
Declarations	$D ::= p(x) \text{ :-} G$
Goals	$G ::= c \mid p(x) \mid G \wedge G \mid G \vee G \mid \exists_x G$

Table 4.1: The language Γ_clp

The language Γ_clp is more general than clp for three reasons. First, it is not necessary to assume that $\mathcal{C}$ contains the equality theory. Second, goals can contain disjunction and quantification. We like to have this feature because we think it provides the goals with a nice algebraic structure, which could be very useful, for instance, for developing a theory of equivalence. Third, in clp global variables can occur only in the goal, whereas in Γ_clp they can occur also in the clauses. For instance in the following program

$$\begin{array}{lll} p(x) & \text{:-} & x = f(y) \\ q(x) & \text{:-} & y = a \end{array}$$

y is a global variable: a call of the form $p(x_1) \wedge q(x_2)$ will link x_1 to $f(a)$.

The second aspect is inherited from ccp paradigm. In particular, we use for disjunction an operator similar to the angelic non-deterministic operator defined in [59]. Here the angelicity of non-determinism consists in allowing all the non-deterministic sub-agents to proceed, and then discarding those which lead to a failure. In clp this corresponds to the mechanism of *consistency check* used to control the choice among the alternative clauses: a branch, whose first action would add a constraint inconsistent with the store, is disregarded. However, another basic primitive (the *ask* operator) is present in (angelic) ccp which allows to enforce synchronization among processes by means of an *entailment check*. Such synchronization mechanism, and then the associated operator, is not present in clp, nor in our language.

4.3 Operational semantics

In this section we present the operational semantics of our language from a "process interpretation" point of view. Namely, we regard an atom in the goal as an agent, and a goal as a set of parallel agents which communicate with each other by establishing constraints on the global variables. In this view, a computation corresponds to the evolution of a dynamic network of parallel agents.

We define the operational semantics in the style of SOS ([80]), i.e. by means of a transition system which describes the evolution of the network in a structural way. The configurations are goals, and the transition relation, $\longrightarrow$, represents the computation step.

4.3.1 The problem of the consistency check

In order to avoid the generation of goals with unconsistent constraints, we have to perform an appropriate consistency check. In Γ_clp it is more complicate than in clp, because of the generalized goals containing the $\vee$ construct. We have to define what is the constraint

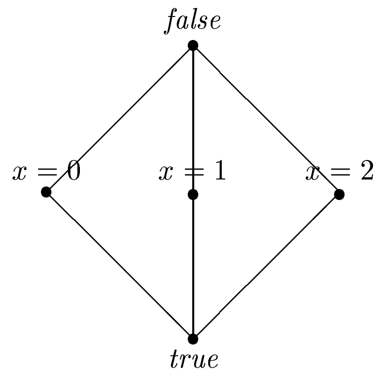

Figure 4.1: A non-distributive constraint system

associated to such goals, and how does it combine with the constraint of a parallel agent. A first choice would be to define the constraint associated to a disjunction as the *greatest lower bound* $\sqcap$ of the two constraints of the disjuncts in the underlying constraint system. However, this choice is not the more appropriate for the consistency check, since it leads to undesirable situations. Intuitively, a goal of the form $G_1 \vee G_2$ will offer both the possibilities of G_1 and G_2. If we put in parallel $G_1 \vee G_2$ with a goal G_3 we can avoid failure if and only if either G_1 or G_2 establish constraints which combine consistently with the ones of G_3. In other words, we need a sort of logical *or*. Unfortunately, the greatest lower bound $\sqcap$ of the two constraints does not correspond to the logical *or*. Consider for example the constraint system illustrated in Figure 4.1. If we have the goal $x = 0 \vee x = 1$, then $x = 0 \sqcap x = 1 \equiv \mathit{true}$, which is consistent with the constraint $x = 2$. On the other hand, $(x = 0 \vee x = 1) \wedge x = 2$ should fail. In order to correspond to the logical and and or, $\sqcup$ and $\sqcap$ should satisfy the distributive laws:

$$\begin{aligned} a \sqcup (b \sqcap c) &= (a \sqcup b) \sqcap (a \sqcup c) \\ a \sqcap (b \sqcup c) &= (a \sqcap b) \sqcup (a \sqcap c). \end{aligned}$$

This is not the case in this example: the lattice in Figure 4.1 is not distributive.

A possible solution could be to impose a priority on the application of the rules of the transition system: disjunctions should be eliminated first. In this way we are sure that, when a consistency check is performed, the goal does not contain disjunctions. Note that this restriction allows us to simulate the efficient linear derivation strategy of logic programming.

However, we adopt another solution which will be useful in the following for defining the denotational semantics, namely we embed the constraint system into a distributive one, where the lub and glb model conjunction and disjunction[2]. A simple way to do this is to lift to sets of constraints, following the idea presented in [28]. In fact, set union and set intersection, which are the glb and the lub on sets, enjoy the distributive property. It turns out that we actually need to consider only sets which are *upward-closed* and *Scott-compact*. We recall that the upward closure of a set $D \subseteq \mathcal{C}$ is the set $\{c \in \mathcal{C} \mid \text{there exists } d \in D.\ d \sqsubseteq c\}$, which we denote

[2]Actually we do not need to have a complete lattice for dealing with the consistency check on the constraints generated during a computation. Since they are always finite, a lattice would be sufficient. However, we will need a complete lattice for the notion of observables (of infinite computations) and the denotational semantics.

by $\uparrow D$. A set $D \subseteq \mathcal{C}$ is *upward-closed* iff $D = \uparrow D$. A *cover* for D is a set of compact elements C such that $D \subseteq \uparrow C$. D is Scott-compact iff every cover for D contains a finite subset which is also a cover. We will denote by $\mathcal{P}_{\mathrm{cu}}(\mathcal{C})$ the set of non-empty Scott-compact upward-closed subsets of $\mathcal{C}$.

Given a cylindric, diagonal constraint system $(\mathcal{C}, \sqsubseteq, \sqcup, \mathit{true}, \mathit{false}, \mathit{Var}, \exists, d)$, consider the structure $\mathbf{C} = (\mathcal{P}_{\mathrm{cu}}(\mathcal{C}), \supseteq, \cap, \mathit{True}, \mathit{False}, \mathit{Var}, \boxminus, \delta)$, where True is $\uparrow \mathit{true}$, which coincides with $\mathcal{C}$, and False is $\uparrow \mathit{false}$, which coincides with $\{\mathit{false}\}$. For $C \in \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$, $\boxminus_x C$ is defined as

$$\boxminus_x C = \{d \mid \text{ there exists } c \in C \text{ such that } \exists_x d = \exists_x c\}.$$

Finally, δ_{xy} is defined as $\uparrow d_{xy}$.
The operator $\boxminus$ enjoys some nice properties, shown by the following propositions.

Proposition 4.3.1 *Let $C \subseteq \mathcal{C}$ be upward-closed. Then*

$$\boxminus_x C = \uparrow \exists_x C,$$

where $\exists_x C$ stands for the pointwise application of $\exists_x$ to the elements of C, i.e. $\exists_x C = \{\exists_x c \mid c \in C\}$.

Proof

($\subseteq$) Let $d \in \boxminus_x C$. Then, there exists $c \in C$ such that $\exists_x d = \exists_x c$. We have

$$d \sqsupseteq \exists_x d = \exists_x c \in \exists_x C,$$

and then $d \in \uparrow \exists_x C$.

($\supseteq$) Let $d \in \exists_x C$. Then, $d \sqsupseteq \exists_x c$, for some $c \in C$. Consider $c' = c \sqcup \exists_x d$. Since C is upward-closed and $c' \sqsupseteq c$, we have that $c' \in C$. By axiom (iii) of Section 4.2.1, $\exists_x c' = \exists_x c \sqcup \exists_x d$, and by axioms (ii) and (v) of Section 4.2.1, $\exists_x d \sqsupseteq \exists_x c$. Thus we have

$$\exists_x c' = \exists_x c \sqcup \exists_x d = \exists_x d,$$

and then $d \in \boxminus_x C$.

$\square$

Proposition 4.3.2 *$\boxminus$ is idempotent, i.e. for every $A \in \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$ and $x \in \mathit{Var}$,*

$$\boxminus_x \boxminus_x A = \boxminus_x A$$

holds.

Proof

$$\begin{aligned}
\boxminus_x \boxminus_x A &= \{d \mid \text{ there exists } c \in \boxminus_x A \text{ such that } \exists_x c = \exists_x d\} \\
&= \{d \mid \text{ there exist } a \in A \text{ and } c \in \mathcal{C} \text{ such that } \exists_x a = \exists_x c \text{ and } \exists_x c = \exists_x d\} \\
&= \{d \mid \text{ there exists } a \in A \text{ such that } \exists_x a = \exists_x d\} \\
&= \boxminus_x A.
\end{aligned}$$

$\square$

Proposition 4.3.3 *$\exists\!\!\exists_x$ is continuous on $\mathcal{C}$, i.e.*

$$\exists\!\!\exists_x \bigcap_i S_i = \bigcap_i \exists\!\!\exists_x S_i,$$

for all families $\{S_i\}_i \subseteq \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$.

Proof

$(\subseteq)$ $\quad d \in \exists\!\!\exists_x \bigcap_i S_i$

$\Rightarrow$

there exists $c \in \bigcap_i S_i$ such that $\exists_x c = \exists_x d$

$\Rightarrow$

there exists $c \in S_i$, for all i, such that $\exists_x c = \exists_x d$

$\Rightarrow$

$d \in \exists\!\!\exists_x S_i$, for all i

$\Leftrightarrow$

$d \in \bigcap_i \exists\!\!\exists_x S_i$.

$(\supseteq)$ $\quad d \in \bigcap_i \exists\!\!\exists_x S_i$

$\Rightarrow$

$d \in \exists\!\!\exists_x S_i$, for all i

$\Rightarrow$

for all i there exists $c_i \in S_i$, such that $\exists_x c_i = \exists_x d$.

Let $c = \bigsqcup_i c_i$. We have

(1) $c \in \bigcap_i S_i$;
(2) $\exists_x c = \exists_x d$.

In fact, $c \sqsupseteq c_i$, and S_i upward-closed for all i, imply (1). For (2), observe that, by continuity of $\exists_x$ over $\mathcal{C}$, the following holds:

$$\exists_x c = \exists_x \bigsqcup_i c_i = \bigsqcup_i \exists_x c_i = \bigsqcup_i \exists_x d = \exists_x d.$$

Therefore, $d \in \exists\!\!\exists_x \bigcap_i S_i$.

$\square$

Proposition 4.3.4 *Let $S \in \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$. Then $\exists\!\!\exists_x S \in \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$.*

Proof Since continuous functions preserve Scott-compactness, by Proposition 4.3.3 we have that $\exists\!\!\exists_x S$ is Scott-compact. Furthermore, it is non-empty, since it contains at least S, which is non-empty by hypothesis. Now we show that it is also upward-closed.

Let $d \in \exists\!\!\exists_x S$ and $d' \geq d$. Let $c \in S$ be such that $\exists_x c = \exists_x d$. Consider $c' = c \sqcup \exists_x d'$. We have that

- $c' \in S$, because $c' \sqsupseteq c$ and, by inductive hypothesis, S is upward-closed. Moreover,
- $\exists_x c'$

  $=$

  $\exists_x (c \sqcup \exists_x d')$

  $=$ $\quad$ { by axiom (iii) in Section 4.2.1}

  $\exists_x c \sqcup \exists_x d'$.

Now observe that, by axiom (ii) in Section 4.2.1, $d' \sqsupseteq d$ implies $\exists_x d' \sqsupseteq \exists_x d$. Then, we have

$$\exists_x c \sqcup \exists_x d' = \exists_x d \sqcup \exists_x d' = \exists_x d'.$$

$\square$

Proposition 4.3.5 *$\boxplus_x$ is downward continuous on $\mathcal{C}$, i.e.*

$$\boxplus_x \bigcup_i S_i = \bigcup_i \boxplus_x S_i,$$

for all families $\{S_i\}_i \subseteq \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$.

Proof

$d \in \boxplus_x \bigcup_i S_i$

$\Leftrightarrow$

there exists $c \in \bigcup_i S_i$ such that $\exists_x d = \exists_x c$

$\Leftrightarrow$

for some i there exist $c \in S_i$ such that $\exists_x d = \exists_x c$

$\Leftrightarrow$

for some i, $c \in \boxplus_x S_i$

$\Leftrightarrow$

$c \in \bigcup_i \boxplus_x S_i$.

$\square$

The correspondence between the original constraint system and the structure **C** is obtained by mapping each element c into $\uparrow c$. In fact, we have:

Proposition 4.3.6 *For all $c \in \mathcal{C}$, $\uparrow c \in \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$.*

Proof It is straightforward that $\uparrow c$ is non-empty and upward-closed for all $c \in \mathcal{C}$. It remains to show that it is also Scott-compact.

Let V be a cover for $\uparrow c$, i.e. $\uparrow c \subseteq \uparrow V$. Since $c \in \uparrow V$, $c \sqsupseteq v$ for some $v \in V$. Therefore, $\{v\}$ is a finite subset of V which is a cover for $\uparrow c$.

$\square$

The mapping $\uparrow : \mathcal{C} \to \mathbf{C}$ preserves the ordering, the lub, and the existential operator. Of course, it does not preserve the glb.

Proposition 4.3.7 *Let $c, d \in \mathcal{C}$. Then*

1. $c \sqsubseteq d \Leftrightarrow \uparrow c \supseteq \uparrow d$;
2. $\uparrow(c \sqcup d) = \uparrow c \cap \uparrow d$;
3. $\uparrow \exists_x c = \boxplus_x \uparrow c$.

Proof

1. $c \sqsubseteq d$

$\Leftrightarrow$

$\forall d' (d' \sqsupseteq d \Rightarrow c \sqsubseteq d')$

$\Leftrightarrow$

$\uparrow c \supseteq \uparrow d$.

2. $d' \sqsupseteq c \sqcup d$

$\Leftrightarrow$

$d' \sqsupseteq d$ and $d' \sqsupseteq c$

$\Leftrightarrow$

$d' \in \uparrow c \cap \uparrow d$.

3.
- $d' \in \uparrow \exists_x c \Rightarrow d' \sqsupseteq \exists_x c$. By (ii) and (v) of Section 4.2.1, $\exists_x d' \sqsupseteq \exists_x c$ holds. Then, for $c' = c \sqcup \exists_x d'$, we have that $c' \in \uparrow c$, and $\exists_x c' = \exists_x c \sqcup \exists_x d' = \exists_x d'$, i.e. $d' \in \exists\!\!\!\exists_x \uparrow c$.
- $d' \in \exists\!\!\!\exists_x \uparrow c \Rightarrow$ there exists $c' \sqsupseteq c$ such that $\exists_x c' = \exists_x d'$. Then, by (ii) we have $d' \sqsupseteq \exists_x d' = \exists_x c' \sqsupseteq \exists_x c$, i.e. $d' \in \uparrow \exists_x c$.

$\square$

Indeed, since $\mathcal{C}$ is a complete lattice, the second point of this proposition can be shown for any subset of $\mathcal{C}$. By an immediate generalization of point 2, we have:

Proposition 4.3.8 *Let $D \subseteq \mathcal{C}$. Then*

$$\uparrow \bigsqcup D = \bigcap_{d \in D} \uparrow d.$$

The following example shows that $\uparrow (c \sqcap d) = \uparrow c \cup \uparrow d$ does not hold, in general.

Example 4.3.9 *Let $\mathcal{C}$ be the constraint system in Figure 4.1, and take $c \equiv x = 1$ and $d \equiv x = 2$.*

The structure $\mathbf{C}$ satisfies the property of being a cylindric, diagonal constraint system. In order to see this, we first show some preliminar results.

Lemma 4.3.10 *Let $A, B \subseteq \mathcal{C}$ be two non-empty, upward-closed sets. Then*

1. $\uparrow minA = A$;
2. $A \cap B = \uparrow \{a \sqcup b \mid a \in minA \text{ and } b \in minB\}$;
3. $\exists\!\!\!\exists_x A = \uparrow \exists\!\!\!\exists_x minA$,

where, for a given set X, $minX$ stands for the set of the minimal elements *of X, $minX = \{x \in X \mid \forall y \in X.\, y \leq x \Rightarrow y = x\}$.*

Proof

1. $c \in A$

$\Leftrightarrow$ $\{A$ is upward-closed$\}$

$c \in minA$, or $c \sqsupseteq a, a \in minA$

$\Leftrightarrow$

$c \in \uparrow minA$.

2. $c \in A \cap B$

$\Leftrightarrow$

$c \in A$ and $c \in B$

$\Leftrightarrow$ $\{$point 1.$\}$

there exist $a \in minA$ and $b \in minB$ such that $c \sqsupseteq a$ and $c \sqsupseteq b$

$\Leftrightarrow$

there exist $a \in minA$ and $b \in minB$ such that $c \sqsupseteq a \sqcup b$

$\Leftrightarrow$

$c \in \uparrow \{a \sqcup b \mid a \in minA$ and $b \in minB\}$.

3. ($\subseteq$) Let $d \in \exists\!\!\!\exists_x A$. Then, there exists $a \in A$ such that $\exists_x d = \exists_x a$. By point 1. and axiom (ii) of Section 4.2.1, we have that for some $m \in minA$, $\exists_x d \sqsupseteq \exists_x m$. Thus, $d \sqsupseteq \exists_x d \sqsupseteq \exists_x m \in \exists\!\!\!\exists_x minA$, and then $d \in \uparrow \exists\!\!\!\exists_x minA$.

   ($\supseteq$) Let $d \in \uparrow \exists\!\!\!\exists_x minA$. Then, $d \sqsupseteq d'$ for some d' such that $\exists_x d' = \exists_x m$ for some $m \in minA$. Consider $c = m \sqcup \exists_x d$. Since $c \sqsupseteq m$, $c \in A$ holds. Moreover, it is easy to see that $\exists_x c = \exists_x m \sqcup \exists_x d = \exists_x d$ holds too. Therefore, $d \in \exists\!\!\!\exists_x A$.

□

Remark 4.3.11 *$(\mathcal{P}_{\rm cu}(\mathcal{C}), \supseteq)$ is just the Smyth powerdomain of the constraint system $\mathcal{C}$ ([97]), under the following algebraic characterization given in [81] for a generic ω-algebraic cpo*[3]*:*
Let D be an ω-algebraic cpo. Then, the Smyth powerdomain of D is

$$(\{X \subseteq D \mid X \textit{ is non-empty, upward-closed, Scott-compact}\}, \supseteq).$$

Now, we are ready to prove the following fundamental proposition.

Proposition 4.3.12 **C** *is a cylindric, diagonal constraint system with finite lub and glb coinciding with set intersection and set union respectively. Furthermore, if $\{C_i\}_i$ is an increasing chain (w.r.t. $\supseteq$) then $lub_i C_i = \bigcap_i C_i$.*

Proof For the first part, we have to prove that:

(a) $(\mathcal{P}_{\rm cu}(\mathcal{C}), \supseteq, \cap, \mathit{True}, \mathit{False})$ is a complete algebraic lattice where *True* and *False* are respectively the least and the greatest element;

(b) For every $C, D \in \mathcal{P}_{\rm cu}(\mathcal{C})$ the following axioms hold:

   (i) $\exists\!\!\!\exists_x C \supseteq C$,
   (ii) if $C \supseteq D$ then $\exists\!\!\!\exists_x C \supseteq \exists\!\!\!\exists_x D$,
   (iii) $\exists\!\!\!\exists_x (C \cap \exists\!\!\!\exists_x D) = \exists\!\!\!\exists_x C \cap \exists\!\!\!\exists_x D$,
   (iv) $\exists\!\!\!\exists_x \exists\!\!\!\exists_y C = \exists\!\!\!\exists_y \exists\!\!\!\exists_x C$.

(c) For every $C \in \mathcal{P}_{\rm cu}(\mathcal{C})$ and $x, y, z \in \mathit{Var}$, the following axioms hold:

   (1) $\delta_{xx} = \mathit{True}$,
   (2) if $z \neq x, y$ then $\delta_{xy} = \exists\!\!\!\exists_z (\delta_{xz} \cap \delta_{zy})$,
   (3) if $x \neq y$ then $C \supseteq \delta_{xy} \cap \exists\!\!\!\exists_x (C \cap \delta_{xy})$.

In the following we show that all the previous points hold.

(a) It is easy to see that $(\mathcal{P}_{\rm cu}(\mathcal{C}), \supseteq, \cap, \mathit{True}, \mathit{False})$ is a lattice where *True* and *False* are respectively the bottom and the top. Therefore, in order to see that it is complete, it suffices to show that for every subset $S \subseteq \mathcal{P}_{\rm cu}(\mathcal{C})$, there exists a *lub*, and $lubS \in \mathcal{P}_{\rm cu}(\mathcal{C})$. Since intersections of upward-closed, Scott-compact sets are again upward-closed, Scott-compact sets (see [47] or [68]), we can immediately realize that $\cap S$ is the element which satisfies the above requirements.

[3]We recall that an ω-algebraic cpo is an algebraic cpo where every ω-chain of elements has a lub

Finally, we show algebraicity. To this end we observe that by Remark 4.3.11 the algebraic elements of $\mathbf{C}$ are the finite non-empty subsets of $K(\mathcal{C})$ ([111]). We denote by $K(\mathbf{C})$ the family of such subsets . Then, we have to prove that for every $C \in \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$,

$$C = \bigcap\{A \in K(\mathbf{C}) \mid A \supseteq C\}.$$

Let us call $\mathcal{A}_C$ the set $\{A \in K(\mathbf{C}) \mid A \supseteq C\}$. Obviously, $\bigcap \mathcal{A}_C \supseteq C$ holds. $\bigcap \mathcal{A}_C \subseteq C$ holds too. In fact, suppose by contradiction that there exists $c \in \bigcap \mathcal{A}_C$ such that $c \notin C$. Then, by point 1. of Lemma 4.3.10, we have that for every $m \in minC$, $c \not\sqsupseteq m$. By recalling that $\mathcal{C}$ is an algebraic (complete) lattice, we know that $m = \bigsqcup\{a \in K(\mathcal{C}) \mid a \sqsubseteq m\}$. Since $c \not\sqsupseteq m$, there must be an element $a_m \in \{a \in K(\mathcal{C}) \mid a \sqsubseteq m\}$ such that $c \not\sqsupseteq a_m$. Then, consider the set $L = \{a_m \mid m \in minC\}$. We have that $c \notin\uparrow L$ (since $\forall m.\ c \not\sqsupseteq a_m$), and $C \subseteq\uparrow L$ (since $\forall m.\ (a_m \sqsubseteq m \Rightarrow\uparrow a_m \supseteq\uparrow m)$), i.e. L is a cover for C. Then, by compactness, there exists $F \subseteq L$, F finite, such that $C \subseteq\uparrow F$. But then $\uparrow F \in \mathcal{A}_C$, and thus, by hypothesis, $c \in\uparrow F$ should hold. This gives the contradiction because $c \notin\uparrow L$ implies $c \notin\uparrow F$.

(b) (i) Obvious, by definition of $\exists_x C$.

(ii) By monotonicity of the upward closure operation.

(iii) By Proposition 4.3.1 it suffices to show that

$$\uparrow\exists_x(C\cap\uparrow\exists_x D) =\uparrow\exists_x C\cap\uparrow\exists_x D.$$

($\subseteq$) Let $a \in\uparrow \exists_x(C\cap \uparrow \exists_x D)$. Then there exists $b \in (C\cap \uparrow \exists_x D)$ such that $a \sqsupseteq \exists_x b$. Since $\exists_x b \in \exists_x C$ and $\exists_x b \in \exists_x(\uparrow \exists_x D)$, we have that $a \in\uparrow \exists_x C$ and $a \in\uparrow \exists_x(\uparrow \exists_x D)$. By Proposition 4.3.1 and Lemma 4.3.2 $a \in\uparrow \exists_x D$ and then $a \in\uparrow\exists_x C\cap\uparrow\exists_x D$.

($\supseteq$) Let $a \in\uparrow\exists_x C\cap\uparrow\exists_x D$. Then there exist $c \in C$ and $d \in D$ such that $a \sqsupseteq \exists_x c$ and $a \sqsupseteq \exists_x d$. Therefore, $a \sqsupseteq \exists_x c \sqcup \exists_x d$. By axiom (iii) of Section 4.2.1, $\exists_x c \sqcup \exists_x d = \exists_x(c \sqcup \exists_x d)$. Since $c \sqcup \exists_x d \sqsupseteq c$ and $c \sqcup \exists_x d \sqsupseteq \exists_x d$, and C and $\uparrow\exists_x D$ are upward-closed, we have that $c\sqcup\exists_x d \in C\cap\uparrow\exists_x D$. Therefore, $\exists_x(c\sqcup\exists_x d) \in \exists_x(C\cap\uparrow\exists_x D)$, and $a \in\uparrow\exists_x(C\cap\uparrow\exists_x D)$.

(iv) We show that

$$\uparrow\exists_x(\uparrow\exists_y C) =\uparrow\exists_y(\uparrow\exists_x C).$$

The assert will follow by Proposition 4.3.1.

$$\begin{array}{l} d \in\uparrow\exists_x(\uparrow\exists_y C) \\ \Leftrightarrow \\ \quad \text{there exists } c \in C \text{ such that } d \sqsupseteq \exists_x m \\ \quad\ \text{for some } m \sqsupseteq \exists_y c \\ \Leftrightarrow \\ d \sqsupseteq \exists_x\exists_y c \\ \Leftrightarrow \quad \{\text{ axiom (iv) of Section 4.2.1 }\} \\ d \sqsupseteq \exists_y\exists_x c. \end{array}$$

Since $\exists_y\exists_x c \in \exists_y(\uparrow\exists_y C)$, we have that $d \in\uparrow\exists_y(\uparrow\exists_x C)$.

(c) (1) By axiom 1. of Section 4.2.1, we have

$$\delta_{xx} =\uparrow d_{xx} =\uparrow true = True.$$

(2)
$$\begin{array}{l} \delta_{xy} \\ = \\ \uparrow d_{xy} \\ = \quad \{ \text{ axiom (ii) of Section 4.2.1 } \} \\ \uparrow \exists_z (d_{xz} \sqcup d_{zy}) \\ = \quad \{ \text{ Proposition 4.3.7 (3.) } \} \\ \boxminus_z \uparrow (d_{xz} \sqcup d_{zy}) \\ = \quad \{ \text{ Proposition 4.3.7 (2.) } \} \\ \boxminus_z (\uparrow d_{xz} \cap \uparrow d_{zy}) \\ = \\ \boxminus_z (\delta_{xz} \cap \delta_{zy}). \end{array}$$

(3) Let $d \in \delta_{xy} \cap \boxminus_x (C \cap \delta_{xy})$. By definition of δ and Proposition 4.3.1, this is equivalent to saying that $d \in \uparrow d_{xy} \cap \uparrow \exists_x (C \cap \uparrow d_{xy})$. Thus, we have that $d \sqsupseteq d_{xy}$ and $d \sqsupseteq \exists_x m$ for some $m \in C \cap \uparrow d_{xy}$. Since C and $\uparrow d_{xy}$ are upward-closed, by Lemma 4.3.10,

$$C \cap \uparrow d_{xy} = \uparrow \{ c \sqcup d_{xy} \mid c \in minC \}.$$

Thus, there exists $c \in minC$ such that $m \sqsupseteq c \sqcup d_{xy}$, and then $d \sqsupseteq \exists_x (c \sqcup d_{xy})$. Therefore, $d \sqsupseteq d_{xy} \sqcup \exists_x (c \sqcup d_{xy})$. By axiom 3. of Section 4.2.1, since $x \neq y$, $d_{xy} \sqcup \exists_x (c \sqcup d_{xy}) \sqsupseteq c$ holds, so $d \sqsupseteq c$ and then $d \in C$.

The second part of the proposition is immediate. In fact, by set-theoretical properties of $\cap$, it is easy to see that for every i, $C_i \supseteq \bigcap_i C_i$ (i.e. $\bigcap_i C_i$ is an upper bound for $\{C_i\}_i$); moreover, if U is any upper bound for $\{C_i\}_i$, then $\bigcap_i C_i \supseteq U$ (i.e. $\bigcap_i C_i$ is a least upper bound for $\{C_i\}_i$). □

From the second part of Proposition 4.3.12 it follows that

Corollary 4.3.13 *The support lattice of* **C** *is distributive.*

The entailment relation on **C**, i.e. the operational counterpart of the inverse of $\supseteq$, which we will denote by $\Vdash$, can be computed by extending the original entailment relation with the standard logical rules for conjunction and disjunction, plus the axioms for the existential operators and the diagonal elements ($\boxminus_x$ and δ_{xy}) corresponding to the laws of Section 4.2.1.

However, our main concern is proving the consistency of constraint C, i.e. that $C \neq \mathit{False}$, or, equivalently, that $C \nVdash \mathit{False}$. To this purpose, having a complete deduction system for $\Vdash$ is not sufficient, because it does not imply the computability of $\nVdash$. Fortunately, in the case of the consistency check, we have a relatively complete system[4] which is described in Table 4.2.

The idea behind this system is to reduce the expressions to disjunctions of elementary expressions of the form $\uparrow c$, whose consistency we are able to test directly on the underlying constraint system. Note that to make this reduction we use the distributivity of $\cap$ wrt $\cup$, the property stated in point 2. of Proposition 4.3.7, and the property $\boxminus_x \bigcup_i \uparrow c_i = \bigcup_i \uparrow \exists_x c_i$. The latter follows by Proposition 4.3.5 and point 3. of Lemma 4.3.10.

Proposition 4.3.14 (**Relative completeness of** *sat*) *The relation sat inductively defined by the system in Table 4.2 completely describes consistency, i.e. for each element $C \in \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$, we derive $sat(C)$ iff $C \neq \mathit{False}$.*

[4] Namely our system is complete provided that $c \neq \mathit{false}$ is semidecidable (hence decidable). This assumption is customary for the constraint systems used in clp.

$$\frac{C \ = \ \cup_i \uparrow c_i \ \ , \ \ D \ = \ \cup_j \uparrow d_j}{C \cap D \ = \ \cup_{ij} \uparrow (c_i \sqcup d_j)} \qquad \frac{C \ = \ \cup_i \uparrow c_i}{\exists_x C \ = \ \cup_i \uparrow \exists_x c_i}$$

$$\frac{C \ = \ \cup_i \uparrow c_i \ \ , \ \ D \ = \ \cup_j \uparrow d_j}{C \cup D \ = \ (\cup_i \uparrow c_i) \cup (\cup_j \uparrow d_j)} \qquad \frac{}{sat(\uparrow c)} \ \ c \neq \mathit{false}$$

$$\frac{C \ = \ \cup_i \uparrow c_i \ \ , \ \ \exists j : \ sat(\uparrow c_j)}{sat(C)}$$

Table 4.2: Deduction system for the consistency check. *sat* stands for "satisfiable", i.e. consistent.

We can now define the function $con : Goals \to \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$ which returns the constraint established by a goal.

Definition 4.3.15
$$\begin{array}{lcl} con(c) & = & \uparrow c \\ con(p(x)) & = & \uparrow true \\ con(G_1 \wedge G_2) & = & con(G_1) \cap con(G_2) \\ con(G_1 \vee G_2) & = & con(G_1) \cup con(G_2) \\ con(\exists_x G) & = & \exists\!\!\exists_x con(G). \end{array}$$

We prove that *con* is well-defined, that is $\forall G,\ con(G) \in \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$.

Proposition 4.3.16 *For all G, $con(G)$ is non-empty, upward-closed and Scott-compact.*

Proof We proceed by induction on the structure of G. The proof is immediate for the basic cases c and $p(x)$. The cases $G_1 \wedge G_2$ and $G_1 \vee G_2$ follow by inductive hypothesis, since finite unions and intersections of non-empty, Scott-compact, upward-closed sets are again non-empty, Scott-compact, upward-closed sets. Finally, the case $\exists_x G$ follows by inductive hypothesis and Proposition 4.3.4. □

Now we have the necessary machinery to define the operational semantics. The consistency check on a goal G consists in verifying $sat(con(G))$.

4.3.2 The transition system

The rules for the transition relation $\longrightarrow$ are defined in Table 4.3. The program $P \equiv D_1.D_2.\cdots.D_q$ is assumed to be fixed.

The execution of a predicate call $p(y)$ is modeled by the recursion rule which replaces $p(y)$ by the body of its definition in the program P, after the link between the actual parameter y and the formal parameter x has been established. Following the method introduced in [89], we express this link by the context $\exists_\alpha(d_{y\alpha} \wedge \exists_x(d_{\alpha x} \wedge \ldots))$, where α is a variable which does not occur in P. Note that through the whole computation only one variable α is needed. This mechanism for treating procedure calls is much more simple and elegant than the machinery of standardization apart used in logic programming.

Disjunction is modeled by taking an arbitrary choice of one of the alternatives which do not bring to inconsistency. There is no need to write explicitly the consistency check, because

Recursion	$p(y) \longrightarrow \exists_\alpha(d_{y\alpha} \wedge \exists_x(d_{\alpha x} \wedge G))$	$p(x) :- G \in P$ $sat(con(G))$
Hiding	$\dfrac{G \longrightarrow G'}{\exists_x G \longrightarrow \exists_x G'}$	
Disjunction	$\dfrac{G_1 \longrightarrow G_1'}{\begin{array}{c} G_1 \vee G_2 \longrightarrow G_1' \\ G_2 \vee G_1 \longrightarrow G_1' \end{array}}$	
Parallelism	$\dfrac{G_1 \longrightarrow G_1'}{\begin{array}{c} G_1 \wedge G_2 \longrightarrow G_1' \wedge G_2 \\ G_2 \wedge G_1 \longrightarrow G_2 \wedge G_1' \end{array}}$	$sat(con(G_1' \wedge G_2))$

Table 4.3: The transition system for Γ_clp.

of the fact that G_1' can be derived already guarantees its consistency. The same applies to the rule of hiding, in fact $con(G) \neq \mathit{false}$ implies $con(\exists_x G) \neq \mathit{false}$.

Parallel composition is modeled as interleaving.

Note that disjunction is the only rule which introduces a logical asymmetry between the antecedent and the subsequent of a computation step, in the sense that the "potential constraint" of one of the two disjuncts is discarded. This means that the observables of a goal will have to be defined in terms of the collection of the results of all computations.

We will use the notations $\longrightarrow^*$ to denote the reflexive and transitive closure of the transition relation $\longrightarrow$, and $\not\longrightarrow$ to indicate the absence of any further transition.

In the following, the class of "terminal" goals, namely those goals of the form $c_1 \wedge \ldots \wedge c_n$ (i.e. consisting of constraints only) will be denoted by *TGoals*.

A computation is *and-fair* iff every goal which occurs in a $\wedge$-context either disappears sooner or later (because of an application of the disjunction rule) or it occurs as the premise in an application of the parallel rule.

4.3.3 The Observables

Following the standard definition, what we *observe* about a goal G in a program P is the set of constraints produced by its computations. The final constraints in case of terminating computations, the limits of intermediate constraints in case of infinite fair computations and *false* when all fair computations fail, namely they get stuck because the consistency check does not allow any further transition. The resulting notion of observables we obtain is similar to the *upward closed observables* defined in [38].

Definition 4.3.17 *Given a program P, for every goal G we define*

$$\begin{array}{rll} \mathcal{O}_P(G) = & \bigcup\{con(G') \mid G \longrightarrow^* G' \textit{ for some } G' \in \textit{TGoals}\} & \\ \cup & \{\textit{false} \mid \textit{ for every and-fair derivation} & \\ & \qquad G \longrightarrow^* G' \not\longrightarrow, \textit{ for some } G' \notin \textit{TGoals}\} & \\ \cup & \bigcup\{\bigcap_n con(G_n) \mid \textit{ there exists an infinite} & \\ & \qquad \textit{and-fair derivation} & \\ & \qquad G_0 = G \longrightarrow \ldots \longrightarrow G_n \longrightarrow \ldots\}. & \end{array}$$

In this definition, $\bigcap_n con(G_n)$ represents the least upper bound of the chain $\{con(G_n)\}_n$ in $\mathbf{C}$. Note that this is not the case in general, because $\bigcap_n con(G_n)$ might be not Scott-compact ($\cap$ is the binary lub, not the lub of infinite sets of elements). However, we work in a (complete) algebraic lattice, where countable intersections of Scott-compact sets are again Scott-compact sets.

Note that we consider a concept of *universal failure* according to the notion of *don't know* non-determinism: a failed computation branch cannot be observed unless all the other branches also fail.

4.4 Denotational semantics

Our aim here is to give a denotational characterization of the constraints computed by a goal.

To this aim we need a compositional definition of the semantic operators corresponding to the syntactic ones, and a least fixpoint of a suitable operator to deal with recursion.

We use the standard technique of the "semantics driven by the syntax" and we define by cases a semantic interpretation function for each syntactic category given in Table 4.1.

As explained in the previous section, the constraint associated to a goal cannot be interpreted in $\mathcal{C}$: we need to consider a distributive structure. Hence the interpretation function will map goals into $\mathcal{P}_{\text{cu}}(\mathcal{C})$. The elements of this domain will be called *processes.*

In order to treat predicate definitions and recursion, we need to introduce the notion of (semantic) environment, namely a function mapping predicate names into processes.

Definition 4.4.1 *Let Pred be a set of predicate symbols. Define*

$$\textit{Env} = \{e \mid e : \textit{Pred} \to \mathcal{P}_{\text{cu}}(\mathcal{C})\},$$

with the ordering $e_1 \preceq e_2$ iff $\forall p.\ e_1(p) \supseteq e_2(p)$.

Since the ordering on *Env* is the pointwise extension of the ordering on a complete lattice, we have:

Proposition 4.4.2 $(\textit{Env}, \preceq)$ *is a complete lattice.*

The bottom $e_\perp$ is defined by

$$\forall p \in \textit{Pred},\ \ e_\perp(p) = \textit{True},$$

and the top $e_\top$ is defined by

$$\forall p \in \textit{Pred},\ \ e_\top(p) = \textit{False}.$$

Programs	$\mathcal{D}[\![\epsilon]\!]e = e$
	$\mathcal{D}[\![D.P]\!]e = \mathcal{D}[\![P]\!](\mathcal{D}[\![D]\!]e)$
	$\mathcal{D}[\![p(x) :- G]\!]e = e[\exists_x(\delta_{\alpha x} \cap \mathcal{G}[\![G]\!]e)/p]$
Goals	$\mathcal{G}[\![c]\!]e = \uparrow c$
	$\mathcal{G}[\![G_1 \wedge G_2]\!]e = \mathcal{G}[\![G_1]\!]e \cap \mathcal{G}[\![G_2]\!]e$
	$\mathcal{G}[\![\exists_x G]\!]e = \exists_x \mathcal{G}[\![G]\!]e$
	$\mathcal{G}[\![G_1 \vee G_2]\!]e = \mathcal{G}[\![G_1]\!]e \cup \mathcal{G}[\![G_2]\!]e$
	$\mathcal{G}[\![p(y)]\!]e = \exists_\alpha(\delta_{y\alpha} \cap e(p))$.

Table 4.4: The interpretation functions $\mathcal{D}$ and $\mathcal{G}$.

For any countable set $E \subseteq Env$ we indicate the least upper bound and the greatest lower bound of E by respectively $\bigcup_{\preceq} E$ and $\bigcap_{\preceq} E$. It is easy to verify that $\bigcup_{\preceq} E$ is defined by

$$\forall p \in Pred.\ (\textstyle\bigcup_{\preceq} E)(p) = \bigcap_{e \in E} e(p),$$

and $\bigcap_{\preceq} E$ is defined by

$$\forall p \in Pred.\ (\textstyle\bigcap_{\preceq} E)(p) = \bigcup_{e \in E} e(p).$$

The equations defining the semantic interpretation functions

$$\mathcal{D} : P \cup D \to [Env \to Env],\ \text{and}\ \mathcal{G} : G \to [Env \to \mathcal{P}_{\mathrm{cu}}(\mathcal{C})]$$

for the denotational semantics are defined in Table 4.4.

4.4.1 Recursion

In order to deal with recursion, we define an operator $\mathcal{D}_P$ on the domain Env and we show that it is continuous. Then we use its least fixpoint to define the processes denoting the goals.

Definition 4.4.3 *On the complete lattice $(Env, \preceq)$, let*

$$\mathcal{D}_P : Env \to Env$$

be the mapping defined by $\mathcal{D}_P(e) = \mathcal{D}(P)e$.

For the sake of simplicity, in the following proofs we assume that the parameter passing in the procedure calls is correctly performed, and we omit the diagonal formulas in their denotations.

Lemma 4.4.4 *Let $e, e' \in Env$. Then, for all G*

if $e \preceq e'$ then $\mathcal{G}[\![G]\!]e \supseteq \mathcal{G}[\![G]\!]e'$.

Proof

$(G = c)$ $\mathcal{G}[\![c]\!]e = \uparrow c = \mathcal{G}[\![c]\!]e'$.

$(G = p(x))$ By hypothesis, we have

$$\mathcal{G}[\![p(x))]\!]e = e(p) \supseteq e'(p) = \mathcal{G}[\![p(x)]\!]e'.$$

$(G = \exists_x G')$ By inductive hypothesis, we have $\mathcal{G}[\![G']\!]e \supseteq \mathcal{G}[\![G']\!]e'$. By Proposition 4.3.3, $\exists\!\!\!\exists_x$ is monotonic; hence $\exists\!\!\!\exists_x \mathcal{G}[\![G]\!]e \supseteq \exists\!\!\!\exists_x \mathcal{G}[\![G]\!]e'$ holds.

The last two cases $G = G_1 \wedge G_2$ and $G = G_1 \vee G_2$ follow immediately by the inductive hypothesis. □

Remark 4.4.5 *Given a program $P = D_1.D_2.\ldots.D_n$ and an environment e, the construction of $\mathcal{D}_P(e)$ is accomplished by n steps which can be inductively described as follows:*

$$\begin{array}{rcl} e^0 & = & e \\ e^i & = & e^{i-1}[\mathcal{G}[\![G_i]\!]e^{i-1}/q_i], \end{array}$$

where q_i is the predicate symbol whose definition in P is $D_i = q_i(x) :\!-G_i$.

Note that $\mathcal{D}_P(e)$ is nothing else that e^n in this construction.

Lemma 4.4.6 *Let $e, e' \in Env$ and let $P = D_1.D_2.\ldots.D_n$. Then*

$$\textit{if } e \preceq e' \textit{ then } e^i \preceq e'^i$$

for all $i \in [0, n]$.

Proof (By induction)

$$\begin{array}{ll} i = 0) & e^0 = e \preceq e' = e'^0 \\ i > 0) & e^i = e^{i-1}[\mathcal{G}[\![G_i]\!]e^{i-1}/q_i]. \end{array}$$

By inductive hypothesis and Lemma 4.4.4, $\mathcal{G}[\![G_i]\!]e^{i-1} \supseteq \mathcal{G}[\![G_i]\!]e'^{i-1}$ holds. Therefore, $e^i(q_i) \supseteq e'^i(q_i)$ holds too.

Now, let $j > i$ (if $j < i$, then the inductive hypothesis applies). We have

$$e^i(q_j) = e(q_j) \supseteq e'(q_j) = e'^i(q_j).$$

□

Proposition 4.4.7 *$\mathcal{D}_P$ is monotonic, i. e.*

$$\textit{if } e \preceq e' \textit{ then } \mathcal{D}_P(e) \preceq \mathcal{D}_P(e').$$

Proof By Remark 4.4.5 and Lemma 4.4.6. □

Lemma 4.4.8 *Let $E \subseteq Env$. For all G*

$$\mathcal{G}[\![G]\!]\ (\textstyle\bigcup_{\preceq} E) = \bigcap_{e \in E} \mathcal{G}[\![G]\!]e.$$

Proof

$$
\begin{array}{lll}
(\subseteq) & & \forall e \in E \;\; e \preceq \bigcup_{\preceq} E \\
& \Rightarrow & \{ \text{ Lemma 4.4.4 } \} \\
& & \forall e \in E \text{ and } \forall G, \quad \mathcal{G}[\![G]\!](\bigcup_{\preceq} E) \subseteq \mathcal{G}[\![G]\!]e \\
& \Leftrightarrow & \\
& & \forall G, \quad \mathcal{G}[\![G]\!](\bigcup_{\preceq} E) \subseteq \bigcap_{e \in E} \mathcal{G}[\![G]\!]e.
\end{array}
$$

$(\supseteq)$ (By induction on the structure of G)

$(G = c)$ $\bigcap_{e \in E} \mathcal{G}[\![c]\!]e = \uparrow c = \mathcal{G}[\![c]\!] \bigcup_{\preceq} E$.

$(G = p(x))$ By definition of $\bigcup_{\preceq}$, we have

$$\bigcap_{e \in E} \mathcal{G}[\![p(x)]\!]e = \bigcap_{e \in E} e(p) = (\bigcup_{\preceq} E)(p) = \mathcal{G}[\![p(x)]\!] \; (\bigcup_{\preceq} E).$$

$(G = \exists_x G')$

$$
\begin{array}{rcl}
d \in \bigcap_{e \in E} \mathcal{G}[\![\exists_x G']\!]e & \Rightarrow & \forall e \in E, \;\; d \in \mathcal{G}[\![\exists_x G']\!]e \\
& \Rightarrow & \forall e \in E, \text{ there exists } c_i \in \mathcal{G}[\![G']\!]e \\
& & \text{such that } \exists_x d = \exists_x c_i.
\end{array}
$$

Therefore, for $c = \bigsqcup_i c_i$, we have that $c \in \bigcap_{e \in E} \mathcal{G}[\![G']\!]e$, and, by continuity of $\exists_x$, $\exists_x c = \exists_x \bigsqcup_i c_i = \bigsqcup_i \exists_x c_i = \exists_x d$. Finally, since by the inductive hypothesis $\bigcap_{e \in E} \mathcal{G}[\![G']\!]e \subseteq \mathcal{G}[\![G']\!](\bigcup_{\preceq} E)$, we have that $d \in \mathcal{G}[\![\exists_x G']\!](\bigcup_{\preceq} E)$.

$$
\begin{array}{lll}
(G = G_1 \wedge G_2) & & \bigcap_{e \in E} \mathcal{G}[\![G_1 \wedge G_2]\!]e \\
& = & \\
& & \bigcap_{e \in E} (\mathcal{G}[\![G_1]\!]e \cap \mathcal{G}[\![G_2]\!]e \\
& \subseteq & \\
& & (\bigcap_{e \in E} \mathcal{G}[\![G_1]\!]e) \cap (\bigcap_{e \in E} \mathcal{G}[\![G_2]\!]e) \\
& \subseteq & \quad \{\text{inductive hypothesis}\} \\
& & \mathcal{G}[\![G_1]\!](\bigcup_{\preceq} E) \cap \mathcal{G}[\![G_2]\!](\bigcup_{\preceq} E) \\
& = & \\
& & \mathcal{G}[\![G_1 \wedge G_2]\!](\bigcup_{\preceq} E). \\
(G = G_1 \vee G_2) & & \bigcap_{e \in E} \mathcal{G}[\![G_1 \vee G_2]\!]e \\
& = & \\
& & \bigcap_{e \in E} (\mathcal{G}[\![G_1]\!]e \cup \mathcal{G}[\![G_2]\!]e) \\
& \subseteq & \\
& & (\bigcap_{e \in E} \mathcal{G}[\![G_1]\!]e) \cup (\bigcap_{e \in E} \mathcal{G}[\![G_2]\!]e) \\
& \subseteq & \quad \{\text{inductive hypothesis}\} \\
& & \mathcal{G}[\![G_1]\!](\bigcup_{\preceq} E) \cup \mathcal{G}[\![G_2]\!](\bigcup_{\preceq} E) \\
& = & \\
& & \mathcal{G}[\![G_1 \vee G_2]\!](\bigcup_{\preceq} E).
\end{array}
$$

□

Proposition 4.4.9 *Let P be a program. $\mathcal{D}_P$ is continuous, i.e.*

$$\mathcal{D}_P(\bigcup_{\preceq} E) = \bigcup_{\preceq} \{\mathcal{D}_P(e) \mid e \in E\},$$

for any countable set $E \subseteq Env$.

Proof

Let $p(y) :- G_p$ be the declaration of p in the program P. Then, by definition of $\mathcal{D}_P$ and $\bigcup_{\preceq}$ we have that

$$\forall p \in \mathit{Pred}.\mathcal{D}_P(\bigcup_{\preceq} E))(p) = \mathcal{G}[\![G_p]\!]\bigcup_{\preceq} E,$$

and

$$(\bigcup_{\preceq}\{\mathcal{D}_P(e) \mid e \in E\})(p) = \bigcap_{e\in E}(\mathcal{D}_P(e))(p) = \bigcap_{e\in E}\mathcal{G}[\![G_p]\!]e.$$

Finally, apply Lemma 4.4.8 with $G = G_p$. □

From the above proposition we know that $\mathcal{D}_P$ has a least fixpoint which is obtainable by at most ω applications of $\mathcal{D}_P$ starting from the bottom $e_\perp$. We show the construction.

Definition 4.4.10 *We define the set* $\{e_i\}_i$ *inductively as follows:*

$$\begin{aligned} e_0 &= {\mathcal{D}_P}^0(e_\perp) = e_\perp \\ e_i &= {\mathcal{D}_P}^i(e_\perp) \end{aligned}$$

Proposition 4.4.11 *For all* i, $e_i \preceq e_{i+1}$.

Proof (By induction)

$(i = 0)$ $\quad e_0 = e_\perp \preceq e_1$

$(i > 0)$ $\quad$ By inductive hypothesis, $e_{i-1} \preceq e_i$ holds, and by Proposition 4.4.7, $e_i = \mathcal{D}_P(e_{i-1}) \preceq \mathcal{D}_P(e_i) = e_{i+1}$.

□

Proposition 4.4.12 *Let* P *be a program.* $\mathcal{D}_P$ *has a least fixpoint and* $\mathbf{fix}\mathcal{D}_P = \bigcup_{\preceq}\{e_i\}_i = e_\omega$.

Proof Standard. □

By Proposition 4.4.9 the operator $\mathcal{D}_P$ has a greatest fixpoint too. We show that it is also downward continuous. This means that its greatest fixpoint is obtained by at most ω step starting from the top of the lattice *Env*. In Section 4.4.3 we will use the greatest fixpoint of $\mathcal{D}_P$ to model the results of the successful and terminating derivations of a goal G.

The analogous of Lemma 4.4.8 and Proposition 4.4.9 hold, namely

Lemma 4.4.13 *Let* $E \subseteq \mathit{Env}$. *For all* G

$$\mathcal{G}[\![G]\!](\bigcap_{\preceq} E) = \bigcup_{e\in E}\mathcal{G}[\![G]\!]e.$$

Proposition 4.4.14 *Let* P *be a program.* $\mathcal{D}_P$ *is downward continuous, i.e.*

$$\mathcal{D}_P(\bigcap_{\preceq} E) = \bigcap_{\preceq}\{\mathcal{D}_P(e) \mid e \in E\},$$

for any countable set $E \subseteq \mathit{Env}$.

Let's show the construction of the greatest fixpoint, $\mathit{gfp}(\mathcal{D}_P)$, of $\mathcal{D}_P$.

Definition 4.4.15 *The set $\{\tilde{e_i}\}_i$ is inductively defined as follows:*

$$\begin{array}{rcl}\tilde{e_0} &=& {\mathcal{D}_P}^0(e_\top) = e_\top \\ \tilde{e_i} &=& {\mathcal{D}_P}^i(e_\top)\end{array}$$

Proposition 4.4.16 *For all i, $\tilde{e}_{i+1} \preceq \tilde{e}_i$.*

Proof (By induction)

$(i = 0)$	$\tilde{e_1} \preceq e_\top = \tilde{e_0}$
$(i > 0)$	By inductive hypothesis, $\tilde{e_i} \preceq \tilde{e}_{i-1}$ holds, and by Proposition 4.4.7 $\tilde{e}_{i+1} = \mathcal{D}_P(\tilde{e_i}) \preceq \mathcal{D}_P(\tilde{e}_{i-1}) = \tilde{e}_i$.

□

Proposition 4.4.17 *Let P be a program. $\mathcal{D}_P$ has a greatest fixpoint and $gfp(\mathcal{D}_P) = \bigcap_{\preceq}\{\tilde{e_i}\}_i = \tilde{e}_\omega$.*

Proof Standard. □

4.4.2 The model $\mathcal{G}_P$

Now we are able to define the semantic model for the Γ_clp language.

Definition 4.4.18 *Let p be a program. Then for all G the denotation of G in P is defined by*

$$\mathcal{G}_P[\![G]\!] = \mathcal{G}[\![G]\!]e_\omega.$$

The following theorem ensures us that the semantic function $\mathcal{G}_P$ actually assigns to each goal a process, i.e. an element of $\mathcal{P}_{\mathrm{cu}}(\mathcal{C})$. By Remark 4.3.11, this means that the domain of the denotations of our model is the Smyth powerdomain of the constraint system $\mathcal{C}$ underlying the language.

Definition 4.4.19 *We define a* nus-set *to be a non-empty, upward-closed, Scott-compact set.*

Theorem 4.4.20 *For all G, $\mathcal{G}_P[\![G]\!]$ is a nus-set.*

First we need to establish some preliminary results.

Lemma 4.4.21 *Let $e \in Env$.*

if $e(p)$ is a nus-set for all $p \in Pred$, then $\mathcal{G}[\![G]\!]e$ is a nus-set for all G .

Proof (By induction on the structure of G)

$(G = c)$ Obvious by definition of $\mathcal{G}[\![c]\!]e$.

$(G = p(x))$ We have that $\mathcal{G}[\![p(x)]\!]e = e(p)$, and by hypothesis $e(p)$ is a nus-set.

$(G = \exists_x G')$ $\mathcal{G}[\![\exists_x G']\!]e = \exists_x \mathcal{G}[\![G']\!]e$. By inductive hypothesis $\mathcal{G}[\![G']\!]e$ is a nus-set. Thus $\exists_x \mathcal{G}[\![G']\!]e$ is non-empty and from Proposition 4.3.4 is upward-closed and Scott-compact.

($G = G_1 \wedge G_2$) We have that $\mathcal{G}[\![G_1 \wedge G_2]\!]e = \mathcal{G}[\![G_1]\!]e \cap \mathcal{G}[\![G_2]\!]e$. By the inductive hypothesis, since finite intersections of nus-sets are nus-sets.

($G = G_1 \vee G_2$) By the inductive hypothesis, since finite unions of nus-sets are nus-sets.

□

Proposition 4.4.22 *For all $i \in \omega$,*

$$\forall p \in \mathit{Pred} \;\; e_i(p) \text{ is a nus-set.}$$

Proof (By induction on i)

($i = 0$) Obvious by definition of e_0.

($i > 0$) Let $p \in \mathit{Pred}$ and $p :- G$ the definition of p in P. Then $e_i(p) = (\mathcal{D}_P(e_{i-1}))(p) = \mathcal{G}[\![G]\!]e_{i-1}$. Since by the inductive hypothesis $e_{i-1}(p)$ is a nus-set, we conclude by Lemma 4.4.21 that $\mathcal{G}[\![G]\!]e_{i-1}$ is a nus-set.

□

Corollary 4.4.23 $\forall p \in \mathit{Pred} \;\; e_\omega(p)$ *is a nus-set.*

Proof Observe that $e_\omega(p) = (\bigcup_{\preceq}\{e_i\}_i)(p) = \bigcap_i e_i(p)$, and apply Proposition 4.4.22 and the fact that countable intersections of upward-closed Scott-compact sets are again upward-closed Scott-compact sets. □

Now, we are able to prove Theorem 4.4.20.

Proof of Theorem 4.4.20 By Corollary 4.4.23 and Lemma 4.4.21 with $e = e_\omega$. □

4.4.3 The correspondence

The observables $\mathcal{O}_P$ and the model $\mathcal{G}_P$ given above are strictly related. In fact, they coincide.

Theorem 4.4.24 *For every program P and goal G,*

$$\mathcal{G}_P[\![G]\!] = \mathcal{O}_P(G).$$

For the proof we need to show some propositions and technical lemmas. We proceed by first proving correctness ($\mathcal{G}_P[\![G]\!] \supseteq \mathcal{O}_P(G)$) and then completeness ($\mathcal{G}_P[\![G]\!] \subseteq \mathcal{O}_P(G)$).

Correctness

In order to prove correctness we show that, for a given goal G, all the reduction steps in the transition system defined in Table 4.3 preserve the meaning of G, i.e. its semantics according to the equations in Table 4.4.

Lemma 4.4.25 *For all G*

$$\mathcal{G}_P[\![G]\!] = \bigcap_i \mathcal{G}[\![G]\!]e_i.$$

Proof By Lemma 4.4.8 with $E = \{e_i\}_i$. □

Proposition 4.4.26 *Let P be a program, and G a goal. Let $G \longrightarrow G_i$, $i \in [1,k]$ for some $k \geq 1$, be all the one-step transitions from G. Then we have*

$$\mathcal{G}_P[\![G]\!] \supseteq \bigcup_{i\in[1,k]} \mathcal{G}_P[\![G_i]\!].$$

Proof (By induction on the structure of G)

$(G = c)$ There are no transition steps.

$(G = p(x))$ The only transition step from $p(x)$ is $G \longrightarrow G'$, where $p(x) :- G'$ is the declaration of p in P. Then we have

$$\begin{array}{ll} & \mathcal{G}_P[\![p(x)]\!] \\ = & \\ & \bigcap_{i\geq 0} e_i(p) \\ = & \\ & \bigcap_{i\geq 1} \mathcal{G}[\![G']\!]e_{i-1} \\ = & \quad \{\text{Lemma 4.4.25}\} \\ & \mathcal{G}_P[\![G']\!]. \end{array}$$

$(G = \exists_x G_1)$ Suppose that for some $k \geq 1$, $G_1 \longrightarrow G_1^i$, $i \in [1,k]$, are all the transition steps from G_1. Then we have

$$\begin{array}{lll} d \in \bigcup_{i\in[1,k]} \mathcal{G}_P[\![\exists_x G_1^i]\!] & \Leftrightarrow & \\ & & d \in \mathcal{G}_P[\![\exists_x G_1^i]\!], \text{ for some } i \in [1,k] \\ & \Leftrightarrow & \\ & & \text{there exists } c \in \mathcal{G}_P[\![G_1^i]\!], \text{ for some} \\ & & i \in [1,k], \text{ such that } \exists_x c = \exists_x d \\ & \Leftrightarrow & \\ & & \text{there exists } c \in \bigcup_{i\in[1,k]} \mathcal{G}_P[\![G_1^i]\!] \\ & & \text{such that } \exists_x c = \exists_x d \\ & \Rightarrow & \quad \{\text{inductive hypothesis}\} \\ & & \text{there exists } c \in \mathcal{G}_P[\![G_1]\!] \\ & & \text{such that } \exists_x c = \exists_x d \\ & \Rightarrow & \\ & & d \in \mathcal{G}_P[\![\exists_x G_1]\!]. \end{array}$$

$(G = G_1 \wedge G_2)$ Suppose that for some $k_1, k_2 \geq 1$, $G_1 \longrightarrow G_1^i$, $i \in [1,k_1]$, and $G_2 \longrightarrow G_2^i$, $i \in [1,k_2]$, are all the transition steps respectively from G_1 and from G_2, such that $sat(con(G_1^i \wedge G_2))$ and $sat(con(G_1 \wedge G_2^i))$. Then $G_1 \wedge G_2 \longrightarrow G_1^i \wedge G_2$, $i \in [1,k_1]$, and $G_1 \wedge G_2 \longrightarrow G_1 \wedge G_2^i,\ i \in [1, k_2]$ are all the transition steps from G. We have

$$
\begin{array}{ll}
& \mathcal{G}_P[\![G_1 \wedge G_2]\!] \\
= & \\
& \mathcal{G}_P[\![G_1]\!] \cap \mathcal{G}_P[\![G_2]\!] \\
\supseteq & \quad \{\text{inductive hypothesis}\} \\
& \bigcup_{i\in[1,k_1]} \mathcal{G}_P[\![G_1^i]\!] \cap \bigcup_{i\in[1,k_2]} \mathcal{G}_P[\![G_2^i]\!] \\
= & \\
& \quad ((\bigcup_{i\in[1,k_1]} \mathcal{G}_P[\![G_1^i]\!]) \cap \mathcal{G}_P[\![G_2]\!]) \\
& \quad \cup(\mathcal{G}_P[\![G_1]\!] \cap (\bigcup_{i\in[1,k_2]} \mathcal{G}_P[\![G_2^i]\!])) \\
= & \\
& \bigcup_{i\in[1,k_1]}(\mathcal{G}_P[\![G_1^i]\!] \cap \mathcal{G}_P[\![G_2]\!]) \cup \bigcup_{i\in[1,k_2]}(\mathcal{G}_P[\![G_1]\!] \cap \mathcal{G}_P[\![G_2^i]\!]) \\
= & \\
& \bigcup_{i\in[1,k_1]}(\mathcal{G}_P[\![G_1^i \wedge G_2]\!]) \cup \bigcup_{i\in[1,k_2]}(\mathcal{G}_P[\![G_1 \wedge G_2^i]\!]).
\end{array}
$$

($G = G_1 \vee G_2$) Suppose that $G_1 \longrightarrow G_1^i,\ i \in [1, k_1]$, are all the transition steps from G_1, and $G_2 \longrightarrow G_2^i,\ i \in [1, k_2]$, are all the transition steps from G_2. Then $G_1 \vee G_2 \longrightarrow G_1^i,\ i \in [1, k_1]$, and $G_1 \vee G_2 \longrightarrow G_2^i,\ i \in [1, k_2]$ are all the transition steps from G.

We have

$$
\begin{array}{ll}
& \mathcal{G}_P[\![G_1 \vee G_2]\!] \\
= & \\
& \mathcal{G}_P[\![G_1]\!] \cup \mathcal{G}_P[\![G_2]\!] \\
\supseteq & \quad \{\text{inductive hypothesis}\} \\
& \bigcup_{i\in[1,k_1]} \mathcal{G}_P[\![G_1^i]\!] \cup \bigcup_{i\in[1,k_2]} \mathcal{G}_P[\![G_2^i]\!].
\end{array}
$$

□

Lemma 4.4.27 *Let G be a goal. Then, for all non-failed derivations*

$$G \longrightarrow G_1 \ldots \longrightarrow G_n \longrightarrow^* \ldots,\quad n \geq 1$$

we have that, for all $n \geq 1$

$$con(G_n) \subseteq \mathcal{G}_P[\![G]\!]$$

holds.

Proof(By induction on n)

($n = 1$) Since the derivation is not failed, we have

$$G \longrightarrow G_1,\ \text{with } G_1 \in TGoals.$$

By Proposition 4.4.26, $\mathcal{G}_P[\![G]\!] \supseteq \mathcal{G}_P[\![G_1]\!]$. Since G_1 is of the form $c_1 \wedge \ldots \wedge c_n$, $con(G_1) = \mathcal{G}_P[\![G_1]\!]$.

($n \geq 1$) By monotonicity of the transition system we have that $con(G_{n+1}) \subseteq con(G_n)$. By inductive hypothesis, $con(G_n) \subseteq \mathcal{G}_P[\![G]\!]$ holds, and then also $con(G_{n+1}) \subseteq \mathcal{G}_P[\![G]\!]$ holds.

□

Theorem 4.4.28 (Correctness)

$$\mathcal{O}_P(G) \subseteq \mathcal{G}_P[\![G]\!].$$

Proof Let $c \in \mathcal{O}_P(G)$. If $c = \mathit{false}$, then the assert follows by the upward closeness of $\mathcal{G}_P[\![G]\!]$. Suppose $c \neq \mathit{false}$. We have two cases;

1. $c \in \mathit{con}(G')$ for some $G' \in \mathit{TGoals}$ such that $G \longrightarrow^* G'$. In this case, Lemma 4.4.27 applies.

2. $c \in \bigcap_n \mathit{con}(G_n)$ for some $\{G_n\}$ such that there exists an infinite and-fair derivation $G_0 = G \longrightarrow G_1 \longrightarrow \ldots \longrightarrow G_n \longrightarrow \ldots$.

   Lemma 4.4.27, $\forall n \geq 1$ $\mathit{con}(G_n) \subseteq \mathcal{G}_P(G)$, and then $\bigcap_n \mathit{con}(G_n) \subseteq \mathcal{G}_P(G)$.

□

Completeness

In order to prove completeness, we distinguish the two cases of finite and infinite (non-failed) derivations for a goal G, and define the two corresponding subsets of constraints in $\mathcal{O}_P(G)$, $\mathcal{O}_{Pterm}(G)$ and $\mathcal{O}_{Pinf}(G)$. Analogously, we define the subsets $\mathcal{G}_{Pterm}[\![G]\!]$ and $\mathcal{G}_{Pinf}[\![G]\!]$, respectively of finite and of infinite constraints in $\mathcal{G}_P[\![G]\!]$. We show that $\mathcal{O}_{Pterm} = \mathcal{G}_{Pterm}$ and $\mathcal{O}_{Pinf} = \mathcal{G}_{Pinf}$. The full completeness is obtained by gathering the two separated results.

Definition 4.4.29 *For all G,*

$$\mathcal{G}_{Pterm}[\![G]\!] = \mathcal{G}[\![G]\!]\mathit{gfp}(\mathcal{D}_P).$$

Proposition 4.4.30 *For all G,*

$$\mathcal{G}[\![G]\!]\mathit{gfp}(\mathcal{D}_P) = \bigcup_i \mathcal{G}[\![G]\!]\tilde{e_i}.$$

Proof By Lemma 4.4.13 with $E = \{\tilde{e_i}\}_i$. □

Definition 4.4.31 *For all G,*

$$\begin{array}{rll}\mathcal{O}_{Pterm}(G) = & & \bigcup\{\mathit{con}(G') \mid G \longrightarrow^* G' \text{ for some } G' \in \mathit{TGoals}\} \\ & \cup & \{\mathit{false} \mid \text{ for every and-fair derivation, either} \\ & & \quad G \longrightarrow^* G' \not\longrightarrow, G' \notin \mathit{TGoals}, \text{ or} \\ & & \quad G_0 = G \longrightarrow G_1 \longrightarrow \ldots \longrightarrow G_n \longrightarrow \ldots\}.\end{array}$$

Lemma 4.4.32 *For all G and for all $i \geq 0$, if $c \in \mathcal{G}[\![G]\!]\tilde{e_i}$, $c \neq \mathit{false}$, then there exists a derivation*

$$G \longrightarrow^* G', \text{ such that } G' \in \mathit{TGoals}, \text{ and } c \in \mathit{con}(G').$$

Proof(By induction on i)

$(i = 0)$ (By induction on the structure of G)

($G = c$) For $d \in \mathcal{G}[\![c]\!]\tilde{e_0} = \uparrow c$, consider the derivation $c \longrightarrow^0 c$. We have that $d \in \uparrow c = con(G')$.

($G = p(x)$) By definition of $\tilde{e_0}$, $\mathcal{G}[\![p(x)]\!]\tilde{e_0} = \{false\}$.

($G = \exists_x G'$) Let $c \in \mathcal{G}[\![\exists_x G']\!]\tilde{e_0}$. Then there exists $d \in \mathcal{G}[\![G']\!]\tilde{e_0}$ such that $\exists_x c = \exists_x d$. By inductive hypothesis, there exists a derivation $G' \longrightarrow^* G''$, with $G'' \in TGoals$ and $d \in con(G'')$. Then, $d \in \exists_x con(G'') = con(\exists_x G'')$. Now consider the derivation $\exists_x G' \longrightarrow^* \exists_x G''$, and observe that $\exists_x G'' \in TGoals$.

($G = G_1 \vee G_2$) If $c \in \mathcal{G}[\![G_1 \vee G_2]\!]\tilde{e_0}$, then either $c \in \mathcal{G}[\![G_1]\!]\tilde{e_0}$ or $c \in \mathcal{G}[\![G_2]\!]\tilde{e_0}$. By inductive hypothesis, there exists a derivation either from G_1 or from G_2 satisfying the assert, and this is a possible derivation from $G_1 \vee G_2$.

($G = G_1 \wedge G_2$) If $c \in \mathcal{G}[\![G_1 \wedge G_2]\!]\tilde{e_0}$, then $c \in \mathcal{G}[\![G_1]\!]\tilde{e_0}$ and $c \in \mathcal{G}[\![G_2]\!]\tilde{e_0}$. By inductive hypothesis, there exist derivations $G_i \longrightarrow^{n_i} G'_i$, with $G'_i \in TGoals$, $c \in con(G'_i)$, $n_i \geq 0 \ \ i \in \{1,2\}$. Thus, $c \in con(G'_1) \cap con(G'_2) = con(G'_1 \wedge G'_2)$. Now we show that $sat(G'_1 \wedge G'_2)$ holds. In fact, since $G'_1, G'_2 \in TGolas$, $con(G'_1 \wedge G'_2)$ is of the form $\uparrow c_1 \cap \ldots \cap \uparrow c_n = \uparrow (c_1 \sqcup \ldots \sqcup c_n)$. By hypothesis, we have $false \sqsupset c \sqsupseteq c_1 \sqcup \ldots \sqcup c_n$. Therefore, $G_1 \wedge G_2 \longrightarrow^n G'_1 \wedge G'_2$, with $n = n_1 + n_2$ is the desired derivation.

($i > 0$) We show only the basic cases $G = c$ and $G = p(x)$. The others can be proved by induction on G as in case ($i = 0$).

($G = c$) The same as in case ($i = 0$) with $\tilde{e_0}$ replaced by $\tilde{e_i}$.

($G = p(x)$) Let $p(y) :- G_p$ the declaration of p in P. Then, $\mathcal{G}[\![p(x)]\!]\tilde{e_i} = \tilde{e_i}(p) = \mathcal{G}[\![G_p]\!]\tilde{e_{i-1}}$. By inductive hypothesis on i there exists a derivation $G_p \longrightarrow^* G'$ with $G' \in TGoals$ and $c \in con(G')$. Then, $p(x) \longrightarrow G_p \longrightarrow^* G'$ is the desired derivation from $p(x)$.

$\square$

Theorem 4.4.33 (Completeness for terminating computations) *For all goals G,*

$$\mathcal{G}_{Pterm}[\![G]\!] \subseteq \mathcal{O}_{Pterm}(G).$$

Proof

$$\begin{array}{ll}
 & c \in \mathcal{G}_{Pterm}[\![G]\!] \\
\Leftrightarrow & \quad \{ \text{ Proposition 4.4.30} \} \\
 & c \in \bigcup_i \mathcal{G}[\![G]\!]\tilde{e_i} \\
\Leftrightarrow & \\
 & \exists i . c \in \mathcal{G}[\![G]\!]\tilde{e_i} \\
\Rightarrow & \quad \{ \text{ Lemma 4.4.32 } \} \\
 & G \longrightarrow^* G',\ G' \in TGoals \text{ and } c \in con(G') \\
\Rightarrow & \\
 & c \in \mathcal{O}_{Pterm}(G).
\end{array}$$

$\square$

Now, we show the second part which concerns infinite derivations.

Definition 4.4.34 *For all G,*

$$\mathcal{G}_{Pinf}[\![G]\!] = \mathcal{G}_P[\![G]\!] \setminus \mathcal{G}_{Pterm}[\![G]\!].$$

Definition 4.4.35 *For all* G,

$$\begin{array}{rl} \mathcal{O}_{Pinf}(G) = & \{\bigcap_n con(G_n) \mid \textit{there exists an infinite and-fair derivation} \\ & G_0 = G \longrightarrow G_1 \longrightarrow \ldots \longrightarrow G_n \longrightarrow \ldots\}. \end{array}$$

The following lemma shows some properties of upward-closed sets, concerning their *minimals* (see Section 4.3.1, Lemma 4.3.10) which will be useful in the proof of completeness.

Lemma 4.4.36 *Let* A, B *be two non-empty, upward-closed subsets of* $\mathcal{C}$. *Then the following hold:*

1. $min(A \cup B) \subseteq minA \cup minB$;
2. $min(A \cap B) \subseteq \{a \sqcup b \mid a \in minA, b \in minB\}$

Proof

1.
$$\begin{array}{l}
c \in minA \cup B \\
\Leftrightarrow \\
(c \in A \text{ or } c \in B) \text{ and } \forall y \in A \cup B . y \sqsubseteq c \Rightarrow y = c \\
\Leftrightarrow \\
\begin{array}{lcl} c \in A & & \text{and } (\forall y \in A \cup B . y \sqsubseteq c \Rightarrow y = c) \\ & \text{or} & \\ c \in B & & \text{and } (\forall y \in A \cup B . y \sqsubseteq c \Rightarrow y = c) \end{array} \\
\Rightarrow \\
\begin{array}{lcl} c \in A & & \text{and } (\forall y \in A . y \sqsubseteq c \Rightarrow y = c) \\ & \text{or} & \\ c \in B & & \text{and } (\forall y \in B . y \sqsubseteq c \Rightarrow y = c) \end{array} \\
\Leftrightarrow \\
c \in minA \cup minB.
\end{array}$$

2.
$$\begin{array}{l}
c \in min(A \cap B) \\
\Leftrightarrow \\
\quad c \in A \text{ and } c \in B \text{ and} \\
\quad \forall y \in A \cap B . y \sqsubseteq c \Rightarrow c = y \\
\Leftrightarrow \quad \{\text{Point (1), since } A, B \text{ are upward-closed }\} \\
\quad \text{there exist } a \in minA \text{ and } b \in minB \text{ such that} \\
\quad c \sqsupseteq a \text{ and } c \sqsupseteq b\text{and} \\
\quad \forall y \in A \cap B . y \sqsubseteq c \Rightarrow c = y \\
\Leftrightarrow \\
\quad \text{there exist } a \in minA \text{ and } b \in minB\text{such that} \\
\quad c \sqsupseteq a \sqcup b\text{and} \\
\quad \forall y \in A \cap B . y \sqsubseteq c \Rightarrow c = y
\end{array}$$

Now, observe that $a \sqcup b \in A \cap B$, since $A \cap B$ is upward-closed, and $c \in A \cap B$. Therefore, the last assertion implies that there exist $a \in minA$ and $b \in minB$ such that $c = a \sqcup b$.

□

Lemma 4.4.37 *For all* $i \geq 0$, *if* $c \in min\mathcal{G}[\![G]\!]e_i$, $c \neq false$, *and* $G \notin TGoals$, *then there exists a derivation*

$$G \longrightarrow^j G' \longrightarrow^* \ldots, \quad con(G') =\uparrow c, \text{ and } j \geq i.$$

Proof (By induction on i)

$(i = 0)$ (By induction on the structure of G)

$(G = c)$ In this case $G \in \mathit{TGoals}$.

$(G = p(x))$ $\mathit{min}\mathcal{G}[\![p(x)]\!]e_0 = \{\mathit{true}\}$. Consider the derivation $p(x) \longrightarrow^0 p(x)$.

$(G = \exists_x G')$ $c \in \mathit{min}\mathcal{G}[\![\exists_x G']\!]e_0 \Rightarrow$ there exists $d \in \mathcal{G}[\![G']\!]e_0 . \exists_x c = \exists_x d$.
By inductive hypothesis there exists a derivation $G' \longrightarrow^j G'' \longrightarrow^* \ldots$, with $\mathit{con}(G'') =\uparrow d$, and $j \geq i$. Then consider the derivation $\exists_x G' \longrightarrow^j \exists_x G'' \longrightarrow^* \ldots$. By Proposition 4.3.7, we have that $\mathit{con}(\exists_x G'') = \boxminus_x \mathit{con}(G'') =\uparrow \exists_x d =\uparrow \exists_x c$. Finally, observe that

$\exists_x \exists_x c = \exists_x c = \exists_x d \Rightarrow \exists_x c \in \mathcal{G}[\![\exists_x G']\!]e_0$, and
$c \in \mathit{min}\mathcal{G}[\![\exists_x G']\!]e_0$ and $\exists_x c \sqsubseteq c \Rightarrow c = \exists_x c$.

$(G = G_1 \vee G_2)$ If $c \in \mathit{min}\mathcal{G}[\![G_1 \vee G_2]\!]e_0$, then either $c \in \mathit{min}\mathcal{G}[\![G_1]\!]e_0$ or $c \in \mathit{min}\mathcal{G}[\![G_2]\!]e_0$. By inductive hypothesis, there exists a derivation either from G_1 or from G_2 satisfying the assert, and this is a possible derivation from $G_1 \vee G_2$.

$(G = G_1 \wedge G_2)$ By Lemma 4.4.36, if $c \in \mathit{min}\mathcal{G}[\![G_1 \wedge G_2]\!]e_0$, then there exists $c_1 \in \mathit{min}\mathcal{G}[\![G_1]\!]e_0$ and $c_2 \in \mathit{min}\mathcal{G}[\![G_2]\!]e_0$ such that $c = c_1 \wedge c_2$. By inductive hypothesis, there exist derivations $G_i \longrightarrow^{n_i} G'_i \longrightarrow^* \ldots$, with $\mathit{con}(G'_i) =\uparrow c_i$, $n_i \geq 0$ $i \in \{1, 2\}$. By hypothesis, $c_1 \sqcup c_2 \neq \mathit{false}$ holds. Thus, $G_1 \wedge G_2 \longrightarrow^{n_1+n_2} G'_1 \wedge G'_2 \longrightarrow^* \ldots$ is the desired derivation.

$(i > 0)$ We show only the basic case $G = p(x)$. The others can be proved by induction on G similarly to the case $(i = 0)$.

$(G = p(x))$ Let $p(y) :- G_p$ the declaration of p in P. Then, $c \in \mathit{min}\mathcal{G}[\![p(x)]\!]e_i$ iff $c \in \mathit{mine}_i(p)$ iff $c \in \mathit{min}\mathcal{G}[\![G_p]\!]e_{i-1}$. By inductive hypothesis on i there exists a derivation $G_p \longrightarrow^j G' \longrightarrow^* \ldots$ with $\mathit{con}(G') =\uparrow c$, and $j \geq i-1$. Then, consider the derivation $p(x) \longrightarrow G_p \longrightarrow^j G' \longrightarrow^* \ldots$, of length $j + 1 \geq i$.

□

Lemma 4.4.38 *Let G be a goal. Then,*

1. $G \in \mathit{TGoals} \Leftrightarrow \mathcal{O}_{Pinf}(G) = \emptyset$;
2. $G \in \mathit{TGoals} \Leftrightarrow \mathcal{G}_{Pinf}[\![G]\!] = \emptyset$.

Proof

1. Obvious by Definition 4.4.35.
2. Let $G \in \mathit{TGoals}$. Then $G = c_1 \wedge \ldots \wedge c_n$ for some $c_i \in K(\mathcal{C})$, $i \in [1, n]$, $n < \omega$. Therefore, $\mathit{con}(G) = \mathcal{G}_P[\![G]\!] = \bigcap_{i=1}^n \uparrow c_i$. By Definition 4.4.31 and Theorem 4.4.33, we have

$$\mathcal{G}_P[\![G]\!] = \mathit{con}(G) = \mathcal{O}_{Pterm}(G) = \mathcal{G}_{Pterm}[\![G]\!].$$

Thus, by Definition 4.4.34, $\mathcal{G}_{Pinf}[\![G]\!] = \emptyset$.

$\square$

Lemma 4.4.39 *If $c \in \mathcal{G}_{Pinf}[\![G]\!]$ and $c \neq false$, then $\forall i \geq 0$, $\exists c_i \in min\mathcal{G}[\![G]\!]e_i$ such that $c_i \sqsubseteq c_{i+1}$ and $\bigsqcup_i c_i \sqsubseteq c$.*

Proof

$$\begin{array}{ll} & c \in \mathcal{G}_{Pinf}[\![G]\!] \\ \Rightarrow & \\ & c \in \mathcal{G}_P[\![G]\!] \\ \Rightarrow & \\ & c \in \bigcap_i \mathcal{G}[\![G]\!]e_i \\ \Leftrightarrow & \\ & \forall i\ c \in \mathcal{G}[\![G]\!]e_i \\ \Leftrightarrow & \{\text{Point 1. of Lemma 4.3.10}\} \\ & \forall i \text{ there exists } c_i \in min\mathcal{G}[\![G]\!]e_i \\ & \text{such that } c_i \sqsubseteq c. \end{array}$$

Therefore, $\bigsqcup_i c_i \sqsubseteq c$ holds.

Now observe that since $c \neq false$, $\forall i \geq 0$, $c_i \sqcup c_{i+1} \neq false$ holds. for all $i \geq 0$, $c_i \sqsubseteq c_{i+1}$ holds. Moreover, since by Proposition 4.4.11 and Lemma 4.4.4, $\mathcal{G}[\![G]\!]e_{i+1} \subseteq \mathcal{G}[\![G]\!]e_i$, we have that for all $i \geq 0$

$$\begin{array}{ll} & \uparrow c_{i+1} \subseteq \mathcal{G}[\![G]\!]e_i \\ \Rightarrow & \{c_i \in min(\mathcal{G}[\![G]\!]e_i)\} \\ & c_i \sqsubseteq c_{i+1}. \end{array}$$

$\square$

Theorem 4.4.40 (Completeness for infinite computations) *For all goals G,*

$$\mathcal{G}_{Pinf}[\![G]\!] \subseteq \mathcal{O}_{Pinf}(G).$$

Proof If $G \in TGoals$, then apply Lemma 4.4.38.

If $G \notin TGoals$, then consider $c \in \mathcal{G}_{Pinf}[\![G]\!]$. If $c = false$, then $c \in \mathcal{O}_{Pinf}(G)$ trivially follows by the upward closeness and non-emptiness of $\mathcal{O}_{Pinf}(G)$. Finally, let $c \in \mathcal{G}_{Pinf}[\![G]\!]$ and $c \neq false$. Then by Lemma 4.4.39 there exist $\{c_i\}_i$, such that for all $i \geq 0$, $\uparrow c_{i+1} \subseteq \uparrow c_i$, and $c_i \in min\mathcal{G}[\![G]\!]e_i$. By Lemma 4.4.37 (since $G \notin TGoals$) for all $i \geq 0$ there exist derivations from G of increasing length j_i. Therefore, there exists an infinite derivation from G with lub $l^* = \bigcap_i \uparrow c_i \in \mathcal{O}_{Pinf}(G)$. Now observe that by Lemma 4.4.39 $c \in l^*$. $\square$

4.4.4 $\mathcal{G}_P$ model and C-semantics

The model defined in Sections 4.3 and 4.4 for the language Γ_clp represents the C-semantics for logic programming. In order to see this correspondence we consider only Γ_clp goals of the form $p(x)$. That the transition system defined in Table 4.3 simulates the SLD-resolution is quite immediate. Concerning the notion of observables, some cares need for correctly interpreting them as the operational model for the C-semantics. This is the success set *NESS* as defined in [40], where only terminating successful derivations are considered. So, the correspondence with our observables is restricted to the first part of this set, namely the part related to the answers produced along terminating successful execution paths. Moreover, in our model these answers are the upward closure of the minimal constraints computed by starting from G, whereas, in *NESS* they are always the empty substitution. However, *NESS* is equivalent (see Section 2.1.2) to the upward closure of the set of the atoms obtained by instantiating the corresponding elementary atoms with the minimal substitutions produced in terminating successful computations (aka the S-success set, *CSS*). Thus, by identifying substitutions with equality constraints we obtain the claimed correspondence.

Of course, what we gained by the $\mathcal{G}_P$ model is a characterization also for infinite computations (the third part of the set of the observables defined in Section 4.3.3). Note that the theory upon which this model lies is a symmetrical theory (the operator $\mathcal{D}_P$ is both continuous and downward continuous), according to what we said in Section 3.3.1.

Concerning the failure set, corresponding in our model to the second part of the set of the observables, it is not exactly the *FFI* set introduced in Section 3.2.1 as the negative counterpart of *NESS*. Dealing with constraints instead of substitutions gives us the possibility of having a more "constructive" notion of failure, in a sense which will be made clear in the next Chapter.

4.5 Perspectives for the S-semantics

The last discussion leaves us not quite pleased with our characterization of infinite computations; we pursue a model which is able to capture the S-semantics of (constraint) logic programs. In this section we discuss how we can attain our aim by using essentially the same technique we have proposed for the construction of the $\mathcal{G}_P$ model.

4.5.1 Powerdomain...

As we already pointed out, the characterization of infinite computations presented above is essentially based on the use of the Smith powerdomain of the original constraint system $\mathcal{C}$. Powerdomain construction is analogous to the powerset construction and also fits in with the various constructions on domains which arise naturally when one follows the Scott-Strackey approach to the semantics of programming languages (e.g. sum, product and exponentiation). For this reason, it is more appropriate for languages with non-deterministic and parallel programming features, where the meaning of a program is in general more complex than simply a self-map over some domains of "states": sets of elements instead of single elements are needed to represent the different courses a non-deterministic computation can follow.

A first powerdomain construction preserving the algebraic properties of the original domain under exponentiation and function space (two vital operations in denotational semantics), was introduced by Plotkin ([82]) for a particular category of algebraic domains, namely those which are SFP. This very complicated construction was streamlined by Smyth in [97] which also introduced a "weak" powerdomain relying on a particularly simple theory and yet adequate for most purposes, subsequently called the Smyth powerdomain. A third powerdomain construction is the so-called Hoare powerdomain because it captures the Hoare partial correctness of a non-deterministic program. The basic idea of all of the three method is, roughly, to start from a partial ordered set (poset) $(D \leq)$ and construe the powerset of D as itself a domain (actually a cpo), with a suitable ordering (on sets). There are three natural ways to pre-order the subsets of a poset D. Let us list them. For $A, B \in \mathcal{P}(D)$:

$$\begin{array}{llll}
A \preceq_S B & \Leftrightarrow & \forall b \in B \exists a \in A . a \leq b & \text{(Smyth preordering)} \\
A \preceq_H B & \Leftrightarrow & \forall a \in A \exists b \in B . a \leq b & \text{(Hoare preordering)} \\
A \preceq_{EM} B & \Leftrightarrow & A \preceq_S B \text{ and } A \preceq_H B & \text{(Egli-Milner preordering)}
\end{array}$$

Note that these are only *preorders*; cpo's are obtained by means of the well-known *completion by ideals* of D, where ideals of equivalence classes (w.r.t. the equivalence induced by the preorder) of sets are considered which are ordered by set inclusion. The Plotkin, Smyth, Hoare powerdomains of D are the completion by ideals of the preorderings $\preceq_{EM}$, $\preceq_S$, $\preceq_H$ on $\mathcal{P}(D) \setminus \emptyset$, respectively.

The three kinds of powerdomain correspond to three different ways to consider programs to be equivalent. We illustrate them by an example.

Example 4.5.1 *Consider the* Γ*_clp programs*

$$\begin{array}{lcl} P_1 & = & \{p(x) :\!- x = 1\}; \\ P_2 & = & \{p(x) :\!- x = 1 \vee p(x)\}; \\ P_3 & = & \{p(x) :\!- p(x)\}. \end{array}$$

One view is that none of P_1, P_2, P_3 are equivalent since P_1 always terminates unlike both P_2 and P_3, and P_2 can terminate unlike P_3. Another is that P_1 and P_2 are equivalent but neither P_1 nor P_2 is equivalent to P_3 since P_1 and P_2 give the same set of results, if any. Another is that P_2 and P_3 are equivalent but neither P_2 nor P_3 is equivalent to P_1 since P_2 and P_3 can both (unlike P_1) fail to terminate and so nothing can be guaranteed of either of them. Of the three view, the first is "what actually happens", the second is the partial correctness property of Hoare, the third is the total correctness property of Dijkstra. Again, the Plotkin powerdomain gives semantics for the first, the Hoare powerdomain gives semantics for the second and the Smith powerdomain gives semantics for the third.

Note that this is according to our assertion that the $\mathcal{G}_P$ model is the Smith powerdomain semantics since $p(x)$ has the same denotation in both P_2 and P_3 ($\mathcal{G}_P[\![p(x)]\!] = \mathcal{C}$) but not in P_1.

Our contention is that S-semantics corresponds to the first view of the three above mentioned, and then it can be captured by using a similar technique as in the construction of the $\mathcal{G}_P$ model, but with the Plotkin powerdomain instead of the Smyth powerdomain. This construction should be obtained by using the Egli-Milner ordering (on the quotient set of $\mathcal{P}(\mathcal{C})$ w.r.t. the equivalence induced by $\preceq_{EM}$) for defining the domain of the semantic environments (like *Env*) for a fixpoint operator similar to $\mathcal{D}_P$. Thanks to the algebraic properties of our constraint systems (they are SFP domains), we think that a continuous operator can be defined (based on a suitable interpretation function $\mathcal{G}$ for the goals[5]) by means of which elements of the Plotkin powerdomain can be assigned to goals as their denotations, which contain the *actual* results of the infinite computations of the goals (not their upward closure).

4.5.2 ... or topology?

A notion of observables more adequate than the set defined in Section 4.3.3 for including the success set of the S-semantics, could be obtained by noting that the processes described by the grammar in Table 4.1 are boundedly non-deterministic processes, so that we can regard their computations as the branches of finitely branching trees T. The possible results of a computation form the "frontier" of T (limits along paths of T). By using König's lemma, it can be shown that this frontier is compact ([98]). Therefore, any model for the language which is able to capture the idea of bounded non-determinism and obtain the correspondence with the observables must fulfill this requirement. Note that, for the semantics $\mathcal{G}_P$, this explains the condition for the denotations to be Scott-compact sets. Scott-compactness is a generalization of

[5]For example the goal c should be mapped into $\{c\}$ and $G_1 \wedge G_2$ in a set of the form $\{a \sqcup b \mid a \in \mathcal{G}[\![G_1]\!]$ and $b \in \mathcal{G}[\![G_2]\!]\}$.

a more operational notion, namely the notion of *finitely generable* sets: it can be thought as the condition replacing König's lemma arguments in reasoning with sets instead of trees. Roughly, the idea of a set being finitely generable is that of a set containing all the limits of the possible computations of a (boundedly) non-deterministic process (the frontier of the derivation tree), whereas the idea of Scott-compactness is that of capturing these limits by "upward closing" the finite elements of the computations. Therefore, the last notion is a weaker notion. Actually, there exist sets which are Scott-compact but not finitely generable.

We think that a semantics for infinite computations matching the S-semantics should satisfy the stronger condition of finite generablity.

In [82], Plotkin uses finitely generable sets to describe its powerdomain construction. This characterization is obtained by considering an ordering $\preceq_{EM}{}'$ weaker than $\preceq_{EM}$ and defined as follows:

$$X \preceq_{EM}{}' Y \text{ iff } \forall A \subseteq K(D).(A \text{ finite and } A \preceq_{EM} X) \Rightarrow A \preceq_{EM} Y,$$

where D stands for an ω-algebraic cpo and $K(D)$ for the set of its algebraic elements. The Plotkin powerdomain corresponds to the family of the finitely generable sets modulo the equivalence, $\approx_{EM}$, induced by $\preceq_{EM}{}'$, with the ordering $\preceq_{EM}{}'$. In [81] it is shown that the canonical representatives of these equivalence classes, actually the maximal elements, are the non-empty, convex and Lawson-closed sets ordered by $\preceq_{EM}$, where a convex set is a set X such that

$$X = \{y \mid \exists x, z \in X. x \sqsubseteq y \sqsubseteq z\},$$

and a Lawson closed set is a set which is closed in the Lawson topology of D. Lawson topology is a very interesting topology which has been recently used in connection with the semantics of logic programming in [11]. An extensive treatment of this topology can be found in [47]. Lawson closed sets enjoy the property to be finitely generable sets, so that they can be thought of as playing the same role as the Scott compact sets in the description of the Smyth powerdomain. Therefore, they should be sufficient to give an appropriate description of the computational behavior of programs. In other words, we think that if we take as denotations for our goals Lawson closed sets, then we can obtain the complete correspondence with the observables of the goals (defined as the frontier of the associated derivation tree). Indeed, the addition of the other property (convexity) makes the elements of the Plotkin powerdomain too "big" for exactly characterize the observables of programs.

Example 4.5.2 *Let P_1 be the Γ_clp program which computes the natural numbers and P_2 the program which computes the even natural numbers, namely*

$$\begin{array}{rcl} P_1 & = & \{p(x) :\!- x = 0 \vee \exists_y (x = s(y) \wedge p(y))\}, \;\; \textit{and} \\ P_2 & = & \{p(x) :\!- x = 0 \vee \exists_y (x = s(s(y)) \wedge p(y))\}. \end{array}$$

The observables of P_1 and P_2 should be obviously distinct sets. However, their convex closures coincide.

Our idea is that a model, say $\mathcal{G}_P^S$, able to capture the S-semantics can be defined by means of a bottom up construction *within* the equivalence classes of the Plotkin powerdomain, i.e. by working with particular (and not maximal) elements of these classes (namely the Lawson closed sets) instead of the classes themselves. Of course, this construction is not a fixpoint construction because the Egli-Milner relation is in general only a preorder. Thus the domain formed by the Lawson closed sets with $\preceq_{EM}$ is not in general a cpo. What we would obtain is a topological characterization of the infinite computations for the S-semantics, rather than a genuine denotational (compositional and fixpoint) semantics.

An interesting relationship between this topological characterization and the model $\mathcal{G}_P$ for the Γ_clp language comes out from their respective descriptions. In fact, as shown in [81], for domains which are SFP, (and then for the constraint systems underlying our language) the non-empty upward-closed Scott-compact subsets are exactly the non-empty upward-closed Lawson-closed subsets (and they in turn are just the upward-closed finitely generable subsets). From this identification it follows that for a goal G we have

$$\mathcal{G}_P[\![G]\!] = \uparrow \mathcal{G}_P^S[\![G]\!],$$

which remind us of the already mentioned relationship between the C-semantics and the S-semantics.

Chapter 5

Heyting Negation

In which we introduce negation in the Γ_clp language and we present an intuitionistic characterization of it by using the Heyting algebras theory.

Heyting algebras have been extensively studied in mathematics as their particular properties make them very interesting structures. Among these properties, one of the most important is related to the theory of topological spaces, where in recent years mathematicians have advocated that the attention were shifted to the open sets lattices as the primary objects of study, instead of to the spaces of points (as classically are regarded topological spaces). These lattices are just Heyting algebras; in particular they are complete Heyting algebras (cHa's), the lattices being, indeed complete lattices.

The significance of cHa's to computer sciences was first recognized by Abramsky ([1]) in connection with the study of Stone duality as a mathematical framework unifying various strands of the theory of programming languages. Stone duality (and the concomitant locale theory) has been introduced in [60], and it refers to a categorical correspondence (a duality of categories) between logical, axiomatic systems (locales) and their correspondent topological models. In computer science where models correspond to denotational semantics and axiomatic systems define program logics, Stone duality provides a mathematical explanation of the relationship between the two kinds of semantics. The role of cHa's in this duality is to represent the objects of a category which is opposite to the category of locales (the actual Stone dual of the category of topological spaces) of which they are, in a sense, more concrete versions.

In this thesis we make a different use of cHa's, namely we exploit their internal structure (i.e. the operations and the laws that they obey), which is not evident in the view of cHa's as objects of a category. On the other hand, cHa's are the algebraic theories (models) corresponding to the intuitionistic logic. They have an operation of negation for which the law of excluded middle, $x \vee \neg x = \mathit{true}$, may fail. The definition of such an operation relies on a property that any lattice must satisfy to be a Heyting algebra. Roughly speaking, this property says that there exists a binary operation between the elements of the lattice able to represent the notion of entailment (logical implication) of the logical system as an element of the algebra itself. With more mathematical terms we can say that negation can be defined in a canonical way (there exists a right adjoint to conjunction). The idea is to use such a definition to introduce negation in the Γ_clp language. What we obtain is, on one hand, the possibility to define an operational model more "constructive" than classical negation; on the other hand the denotational semantics for the Γ_clp language introduced in Chapter 4 easily extend to this kind of negation by completely capturing its operational meaning.

In logic programming the problem of providing negation with a more constructive meaning has been tackled by means of various extensions of Negation As Failure, called *constructive negation.*

5.1 Constructive negation

A mathematician who hears of "constructive negation" could be deceived by this name, that, at first, seems to refer to those theories which has been developed (aside the main stream of conventional mathematics) after (and in opposition to) the advent of non-constructive reasoning introduced by Cantor with his diagonalization principle. Before Cantor the restriction to constructive proofs was almost taken for granted, so constructive logic is the original form of mathematical logic. The basic principle of constructivism is rejecting any existence proofs that do not explicitly construct the mathematical object the existence of which is proven. Modern theories preserves this principle, but are very far from the simplicity of their ancestor, for they often use very complicated vocabularies and notations. Fortunately, despite its name, constructive negation is not the negation within one of such theories. In the intentions of its inventor Chan ([17]) it simply indicates an extension of the NAF rule able to return answers for negative subgoals.

Let's briefly motivate the rise of this particular way of handling negation in logic programming.

In Chapter 3 we studied three forms of failure: ground finite failure, non-ground finite failure and failure by instantiation. The associated failure sets are constituted respectively by the ground atoms whose negation is inferred under NAF, the non-ground atoms whose universally closed negation is inferred under NAF, and the non-ground atoms whose existentially closed negation is inferred under NAI. In all cases we meta-logically infer negative statements by "complementing" those positive statements A which cannot be proved (or instantiated) and we specify by means of the existential or universal quantifier that for some or all instantiations of the variables of A its complement, $\neg A$, is true. But none of such instantiations are exhibited as a testimony of the truth of $\neg A$. This fact has been recognized by researchers to be a major drawback of the procedural use of negation as failure in logic programming and various solutions have been proposed aiming at allowing a more symmetric treatment of positive and negative knowledge: negative subgoals would be not used only as a "test" for checking universally or existentially quantified negative formulas, but they would be used as well as positive subgoals to compute answers for existentially quantified negative formulas. This would be useful in many applications such as deductive databases where computing answers for positive and negative queries in a homogeneous way would be a very appealing possibility.

The various extensions of NAF which have been proposed in this direction are grouped under the name of *constructive negation.* By constructive negation "complementation" is performed (in some ways) on the answers returned by a goal A, rather than on A itself, in such a way that an object is explicitly constructed which proves $\neg A$. This is, roughly, the common feature of the various approaches that earns them the attribute of "constructive".

Another approach to the problem of appropriately handling non-ground negative atoms is based on a "transformational" technique, ([10]), and it is known as *intensional negation.* The idea is that of synthesizing the definition of a new predicate $\tilde{p}$ for each predicate p occurring in the program, the intended meaning of $\tilde{p}$ being, roughly speaking, the negation of p. The effect of this transformation is that answers to negative queries are "the complement" of the set of answers to the corresponding positive query; thus intentional negation is indeed (a version of) constructive negation, the only difference being the "compilative" aspect of the former (transformations and normalization processes are performed on the source of the program) contrasting with the "interpretative" aspect of the latter (answer substitutions for negative goals are computed in run-time).

In line with the usual methodology adopted in logic programming for studying the introduction of the negation rules, some approaches to constructive negation have been first developed in the context of positive programs. For example, in [73] a method is presented which, given a definite program P, is able to generate answers (called *fail substitutions*) for negative (existential) queries of the form $\neg(A_1, \ldots, A_n)$. Also the above mentioned transformational method has been initially proposed only for positive programs. Extensions of these methods have been then considered to deal also for general programs. In this context constructive negation has been proposed as a solution to a problem of the interpreter for general programs based on negation as failure (the SLDNF-resolution), namely the problem of floundering. In Chapter 6 we will give some more details on this subject.

5.1.1 Constructive negation and clp

In logic programming answers to positive queries are substitutions, i.e. mappings from variables to terms in the Herbrand Universe, U_H, of the form $\{x_1/t_1, \ldots, x_n/t_n\}$, which can be alternatively (and equivalently) regarded as sets of equalities $\{x_1 = t_1, \ldots, x_n = t_n\}$, thereby looking like to constraints. Actually, logic programming can be viewed as an instantiation of the CLP(X) scheme, namely CLP(U_H). In this view, unifiability of terms corresponds to solvability in U_H of equations between terms.

On the other hand, answers to negative queries are naturally and conveniently expressible by means of inequalities, and there are no ways to represent them by means of substitutions. Therefore, the passage from logic programming to constraint logic programming is quite natural when we have to deal with constructive negation. These considerations have inspired the work of Stuckey, ([102]), in which the clp framework is assumed in its fullest (i.e. constraints are not limited to equations and disequations) in order to give a formalization of constructive negation in the context of general programs.

Another possibility is that of staying within logic programming setting based on substitutions and assuming the Domain Closure Axiom. Informally, this axiom forces the interpretation domain of a model to contain only objects which can be constructed (even not effectively) using the interpretation of constant and function symbols. The technical reason for assuming DCA is that it allows to express the complement of a term (and then an inequality) by means a disjunction of equalities. This is the approach, for example, of the two above mentioned methods ([73] and [10]).

An intermediate (maybe the most natural) solution is that of choosing the instance of clp scheme, where constraints are equalities and inequalities on the Herbrand Universe. This solution has been adopted, for example, in [17] and [85], where, in order to simplify the treatment of equality formulas, the underlying language is assumed (more or less explicitly) to be infinite. In those works the same constructive negation is used within different contexts. In particular, the former has to do with general programs and the completion semantics, whereas the latter has been developed for a particular kind of general programs (the stratified programs) and assumes a semantics "ad hoc" for these programs, (namely th perfect models semantics). As it will be clear in Chapter 6, the two resulting interpretations of negation are different from each other except the property to be constructive.

5.1.2 Γ_clp vs clp

In our treatment of negation we adopt, as in [102], the most general clp setting, where constraints can be anything. However, our approach is different from all the above mentioned approaches for at least two reasons: firstly it is more "algebraic"; secondly, it relies on a semantic model in which also the results of infinite computations are provided with a precise meaning. The algebraicity derives from our use of constraints systems in the style of [88], where the structure of lattice (its lub's, top, bottom, etc., and the associated algebraic properties), rather than some external interpretation structure, plays a fundamental role in defining the meaning of the computations. On the contrary, the way in which domains of constraints are defined in logic programming is typically that of [54] where constraints have a sense whenever interpreted in some (arbitrary) structures which are indispensable for establishing notions such as *satisfiability* and then in determining the meaning of a computation. Note that the notion of satisfiability defined in Chapter 4 is purely syntactic, by relying on a system of axioms and deduction rules. More importantly, the choice of an algebraic approach we will give us the possibility to exploit some well-known results of the theory of lattices in defining the semantic model for negation. An other facility in our treatment of negation comes from disposing of a model for the infinite computations. The typical approaches to constructive negation are based on constructing an answer for a goal $\neg G$ by negating the (disjunction of) the answers computed by the successful derivations for G. To this aim a complete SLD-tree rooted in G should be generated. The difficulty in doing so is that the tree might be infinite. Due to this problem the methods which have been defined for constructive negation are in general more complicated than what above described. On the other hand, constructing a complete derivation-tree is not a problem in our case, since the observables of G include also the infinite constraints (i.e. those constraints which are produced by infinite computations). This gives us the possibility to define a very simple operational model. However, the transition rule for negation we will introduce in the following is not "constructive" according to what we have pointed out at the beginning. For this reason we prefer to call our negation an *intuitionistic* negation, for the model we define is that of the Heyting negation.

5.2 Intuitionistic negation

The identification of logical systems with algebraic theories represents a fairly standard trick used in logic to prove the equivalence between formulas. Thinking of the logical connectives and axioms of logics as the operators and laws of algebras, we can use the methods of algebras to prove equivalences. This process of algebraicizing logics, ([109]), comes in handily in better understanding concepts as posets, lattices, etc. and has its major expression in the theory of *locales* in which also the notions of correctness and completeness of a logical system have their correspondents. As an example of the way in which algebra reflects logic, consider a poset $(P, \leq)$ and think of P as a set of "propositions" (atomic formulas), and of $\leq$ as meaning "entails" ($\vdash$). The properties of reflexivity, transitivity and antisymmetry are the axioms. Lattices have in addition all finite *lub*'s and *glb*'s which are the (algebraic) operators corresponding to the logical connectives $\vee$ (or) and $\wedge$ (and); distributive lattices obey an additional distributive law, and so on: building in more and more logic we see in turn more and more complex structures.

In this framework, (complete) Heyting algebras are the algebraic theories corresponding to the intuitionistic logic.

Definition 5.2.1 *([47]) A* complete Heyting algebra *(cHa) is a complete lattice which satisfies* *the following infinite distributive law:*

$$x \wedge \bigvee Y = \bigvee \{x \wedge y \mid y \in Y\}, \tag{5.1}$$

for all element x and all subsets Y.

The property (5.1) is also called *frame distributivity* to distinguish it from the finite distributivity mentioned in Section 4.3.1. The most typical examples of cHa's are the *topologies.* Given a topological space X, the topology of X is the set of *open subsets* of X. It is a sublattice of the power set of X, 2^X, closed under finite intersections and under arbitrary unions, and it is clearly a cHa since we know the truth of (5.1) for the set-theoretical operations (union and intersection) representing here lub and glb. On the contrary, the set of *closed subsets* of X is not in general a cHa. In fact, it is obviously isomorphic to the opposite of the set of open sets which is not closed under arbitrary intersections.

For generic lattices (without completeness) the property of being Heyting algebras is given by the following definition where the intuitionistic aspect of these structures is more evident.

Definition 5.2.2 *([109]) Let A be a lattice. A is a* Heyting algebra *iff for every $x, y \in A$ there is an element $x \to y$ satisfying*

$$z \leq x \to y \textit{ iff } z \wedge x \leq y. \tag{5.2}$$

A Heyting algebra A is a cHa iff it is a complete lattice.

The lattice theory shows that in a lattice enjoying the property (5.1) it is always possible to uniquely define the operation $\to$ in a way such that (5.2) is satisfied. The standard way to do so is as follows:

Definition 5.2.3 $\quad x \to y = \bigvee \{z \in A \mid x \wedge z \leq y\}$

If a Heyting algebra A has a bottom, then a unary operation, the *Heyting negation* is defined by

Definition 5.2.4 $\quad \neg x = x \to \mathit{false}$[1].

Notice that for the Heyting negation, $\neg \mathit{true} = \mathit{false}$ and $\neg false = \mathit{true}$ hold. Moreover,

$$x \wedge \neg x = \mathit{false} \tag{5.3}$$

always holds, but

$$x \vee \neg x = \mathit{true} \tag{5.4}$$

does not hold, in general.

The last assertion is according to what we said at the beginning of this chapter, because (5.4) is the excluded middle law.

[1]According to the standard interpretation of the logical connectives, constants *true* and *false* correspond respectively to the top element $\top$ and the bottom element $\bot$ of the lattice. Note also that, due to the identification, logical symbols are often used instead of the corresponding algebraic, so, for instance, $a \vee b$ is used for $lub(a, b)$, *true* for $\top$, and so on.

5.2.1 Intuitionistic negation and Γ_clp

We consider now the possibility of introducing in the language defined in Section 4.2.2 a construct for negation. Of course, in doing so, we would like to maintain as much as possible the correspondence between the operational and the denotational model described in Sections 4.3 and 4.4 for the language without negation. In this section we show that this is possible for the "minimal" extension of the Γ_clp language, namely for the language in which negative goals are not allowed to occur in the bodies of the clauses. On the contrary, as we will see in Section 5.2.1, allowing the unconditioned use of negation in the recursive definitions leads to some problems with the continuity of the fixpoint operator, so that the model $\mathcal{G}_P$ does not work any more, without suitable restrictions on the use of the negation construct.

So we aim, for the moment, at having the possibility of computing negative goals, from *positive* programs only. Also we don't consider neither nested negation nor negation inside a $\vee$ context: our goals will be conjunctions of positive and negative goals, and G will always indicates a positive goal as defined in Table 4.1.

In order to model $\neg G$, we have to define, first of all, what is the constraint associated to a negated goal. More in general, we have to extend the structure $\mathbf{C}$ with a notion of negation. The first idea would be to define $\neg C$ as $\mathcal{C} \setminus C$, but for doing this we should extend $\mathbf{C}$ with sets which are not upward closed. Moreover, with this notion we would obtain the classical negation; on the contrary we would like to model at least a form of negation as failure. To this end the choice of the Heyting negation appears to be more appropriate. However, $\mathbf{C}$ is not a cHa because it is not closed under arbitrary unions (they are not Scott-compact), so that the frame distributivity (5.1) does not hold in general. In order to obtain a cHa we consider a structure with support containing all the upward closed subsets of $\mathcal{C}$, not only the Scott-compact ones. Moreover, we assume the ordering to be set inclusion, to adhere the standard semantic recalled in the previous section, (in Section 4.3.1 we have adopted the reverse set inclusion to follow the standard lines in the study of infinite computations, namely to define, for the fixpoint operator $\mathcal{D}_P$, a domain correspondent to the Smyth powerdomain of $\mathcal{C}$). Therefore, we consider $\mathbf{C}'$ to be the structure $(\mathcal{P}_{\mathrm{u}}(\mathcal{C}), \subseteq, \cup, \{\mathit{false}\}, \uparrow \mathit{true}, \mathit{Var}, \exists\!\!\!\exists, \delta)$, where $\mathcal{P}_{\mathrm{u}}(\mathcal{C})$ is the family of the upward closed subsets of $\mathcal{C}$, $\uparrow \mathit{true}$ (i.e. $\mathcal{C}$) is the top, and $\{\mathit{false}\}$ is the bottom. Then, we define $\neg C$ as in Definition 5.2.4, namely

$$\neg C = \bigcup \{X \mid X \cap C = \{\mathit{false}\}\}.$$

An equivalent definition is as follows:

$$\neg C = \{d \mid \forall c \in C.\ c \sqcup d = \mathit{false}\}.$$

The function *con* of Definition 4.3.15 is extended to negative goals by posing

$$\mathit{con}(\neg G) = \neg \mathit{con}(G).$$

Notice that from (5.3) it follows that

$$\mathit{con}(\neg G) \cap \mathit{con}(G) = \{\mathit{false}\}. \tag{5.5}$$

Operational semantics and observables of negative goals

From the point of view of the structural operational semantics, the evolution of $\neg G$ should be determined by the evolution of G. Hence we want to have a rule of the form

$$\textbf{Negation} \quad \frac{G \longrightarrow G'}{\neg G \longrightarrow \neg G'} \quad \mathit{con}(\neg G') \neq \{\mathit{false}\}$$

However this rule, if combined with the notion of observables given before, is unsound. This is due to the asymmetry introduced by the disjunction rule: in general after an application of the disjunction rule we have $con(G') \subseteq con(G)$, therefore after the application of the negation rule we have that $con(\neg G')$ contains more constraints than the "original possibilities" of $\neg G$. Collecting the results like we did before would assign to negation a wrong meaning, as the following example shows.

Example 5.2.5 *Consider the program*

$$p(x) :- x = 0 \vee x = 1,$$

in a system where $true, x = 0, x = 1$ *and* $false$ *are the only constraints. The goal* $p(x)$ *has two possible derivations:*

$$p(x) \longrightarrow^* x = 0 \quad \textit{and} \quad p(x) \longrightarrow^* x = 1.$$

By the negation rule, $\neg p(x)$ *has the derivations:*

$$\neg p(x) \longrightarrow^* \neg(x = 0) \quad \textit{and} \quad \neg p(x) \longrightarrow^* \neg(x = 1).$$

Since $\neg(x = 0)$ *corresponds to* $\{x = 1\}$ *and* $\neg(x = 1)$ *corresponds to* $\{x = 0\}$, *we would conclude that* $p(x)$ *and* $\neg p(x)$ *have the same observables!*

However, the negation rule in itself is not unsound. It only makes necessary to adopt a suitable notion of observables.

As explained above, the negation rule inverts the asymmetry introduced by the disjunction rule. As a consequence, also the way we collect the observables must be dual: instead of the union, we have to take the intersection. We extend therefore the function $\mathcal{O}_P$ on negative goals as follows:

Definition 5.2.6

$$\begin{aligned}
\mathcal{O}_P(\neg G) = \quad & \bigcap\{con(\neg G') \mid \neg G \longrightarrow^* \neg G' \textit{ for some } G' \in TGoals\} \\
\cap \quad & \uparrow\{true \mid \textit{for every and-fair derivation } \neg G \longrightarrow^* \neg G' \not\longrightarrow, G' \notin TGoals\} \\
\cap \quad & \bigcap\{\textstyle\bigcup_n con(\neg G_n) \mid \textit{there exists an infinite and-fair derivation } \neg G_0 = \neg G \longrightarrow \ldots \longrightarrow \neg G_n \longrightarrow \ldots\}.
\end{aligned}$$

This definition of observables seems the most natural from the point of view of the process interpretation; note, however, that it does not correspond to the standard notion of *negation as finite failure* in (constraint) logic programming. To obtain such kind of negation we should include in $\mathcal{O}_P(\neg G)$ only the first two sets, corresponding to the information we can derive from finite computations.

Denotations of negative goals

The natural idea for defining the denotations of negative goals is to assign to $\neg G$ the Heyting negation of the process associated to the corresponding positive goal G, namely to define

$$\mathcal{G}_P[\![\neg G]\!] = \neg \mathcal{G}_P[\![G]\!].$$

As we already said, what we obtain is a set which is still upward closed, but not necessarily Scott-compact. Consequently, the domain of the denotations has been enlarged to $\mathcal{P}_{\mathrm{u}}(\mathcal{C})$. In order to define a denotational semantics for the Γ_clp language extended with the full use of negation (i.e. the language obtained by adding to the grammar of Table 4.1 the production $\neg G$ as a possible alternative for the category of the goals, G) we should dispose of a continuous operator over $\mathcal{P}_{\mathrm{u}}(\mathcal{C})$ similar to the operator $\mathcal{D}_P$ for the positive language. Unfortunately, the operator $\neg$ is not monotonic, thus no semantic operator can be continuous for the language including $\neg$, unless restrictions are imposes on its use. Actually, it is possible to show that conditions similar to the stratification for logic programs, ([5]), are definable also for our language so that the operator $\mathcal{D}_P$ (which is now not monotonic) has a fixpoint by means of which a *natural* model for the goals can be defined. Another possibility is to find what uses of $\neg$ allow to preserve the continuity of $\mathcal{D}_P$. Indeed, it is possible to show that "double negation" has this property. The possibilities offered by a language including such a construct are not equivalent to the positive language because for the Heyting negation $\neg\neg G = G$ does not hold, in general.

For positive programs, the simple extension of the denotational semantics $\mathcal{G}_P$ above defined is a good semantics for negative goals. In fact, the correspondence with the observables is maintained:

Proposition 5.2.7 $\mathcal{G}_P[\![\neg G]\!] = \mathcal{O}_P(\neg G)$.

Proof Distinguish two cases:

($\mathcal{O}_P(\neg G) = \uparrow true$) We have that for every and-fair derivation from G, $G \longrightarrow^* G' \not\longrightarrow$, for some $G' \notin TGoals$, i.e. $\mathcal{O}_P(G) = \{false\}$. Then,

$$\begin{array}{ll} & \mathcal{O}_P(G) = \{false\} \\ \Leftrightarrow & \quad \{\text{ Theorem 4.4.24 }\} \\ & \mathcal{G}_P[\![G]\!] = \{false\} \\ \Leftrightarrow & \quad \{\text{ property of the Heyting negation }\} \\ & \neg\mathcal{G}_P[\![G]\!] = \uparrow true \\ \Leftrightarrow & \\ & \mathcal{G}_P[\![\neg G]\!] = \uparrow true. \end{array}$$

($\mathcal{O}_P(\neg G) \neq \uparrow true$) Let $d \in \mathcal{O}_P(\neg G)$. If $d = false$, the theorem is immediate since both $\mathcal{O}_P(\neg G)$ and $\mathcal{G}_P[\![\neg G]\!]$ are non-empty and upward-closed. Thus, consider $d \neq false$. Let $\longrightarrow^\infty$ indicate the union of $\longrightarrow^*$ and the transitive infinite closure of the transition relation $\longrightarrow$, $\longrightarrow^\omega$. We have

$$\begin{array}{ll} & d \in \mathcal{O}_P(\neg G) \\ \Leftrightarrow & \\ & \uparrow d \in con(\neg G'),\ \forall G' \text{ such that } G \longrightarrow^\infty G' \\ \Leftrightarrow & \quad \{(5.5)\} \\ & \uparrow d \cap con(G') = \{false\},\ \forall G' \text{ such that } G \longrightarrow^\infty G' \\ \Leftrightarrow & \\ & \uparrow d \cap \mathcal{O}_P(G) = \{false\} \\ \Leftrightarrow & \quad \{\text{Theorem 4.4.24}\} \\ & \uparrow d \cap \mathcal{G}_P[\![G]\!] = \{false\} \\ \Leftrightarrow & \\ & \uparrow d \in \neg\mathcal{G}_P[\![G]\!] \\ \Leftrightarrow & \\ & d \in \mathcal{G}_P[\![G]\!]. \end{array}$$

$\square$

Note that this definition of negation has a close correspondence with the set of the *finite failures* of G as defined in [96]:

$$\mathcal{O}_{P,f}(G) \quad = \quad \{c \mid c \wedge G \longrightarrow^{*} \mathit{false}\}.$$

The difference is that in our case we have

$$\mathcal{O}_P(\neg G) \quad = \quad \{c \mid \mathcal{O}_P(c \wedge G) = \{\mathit{false}\}\},$$

which includes the possibility that G has an infinite computation giving *false* as the limit result.

Chapter 6

General programs

In which we consider general programs, and the possible extensions to them of the results obtained for positive programs.

"...As far as negation as failure is concerned, pure Horn clause logic is undoubtedly one of the few cases where its semantics is clear and completely acceptable" ([95]); anyway, in the last years research in logic programming semantics has been constantly trying to develop semantics for (suitable notions of) negation as failure for classes of logic programs larger than definite programs.

In this chapter we shall discuss extensions to *general programs*.

Some more recent research has concerned more general kinds of programs, notably programs with disjunctions of atoms in the heads of the clauses (disjunctive logic programs, see for instance [9]), or logic programs with classical negation ([46]). But we shall not address such programs here, as it is stuff for a separate treatment.

With general programs (sometimes called normal programs) there is no universally accepted semantics for negation as failure, and new proposals keep on proliferating. The various investigations have tended essentially towards two directions.

On one hand, the "procedural" meaning of negation has been assumed as a guide in defining a declarative semantics for general programs, so that either the "classical" (i.e. Prolog-style) meaning of negation as failure has been completely preserved, or other forms of negation have been graft to the very efficient interpreter used in Prolog-style languages (SLD-resolution). In both cases, the resulting semantics are all *completion-based*, in the sense that the intended meaning of a program P is always understood to be the Clark's completion of P, $Comp(P)$, maybe in some extended forms or with an underlying logics other than the classical 2-valued logics.

On the other hand, the treatment of negation has been (by using a term of [103]) *declaratively-driven*, in the sense that the various semantics have been defined in terms of special classes of models for the program before an interpreter was even designed. As a consequence of this fact, the interpretation of negation does not correspond, in general, to the "classical" negation as failure. This line of research is more tied to the realm of Artificial Intelligence, in particular to the area devoted to non-monotonic reasoning. In fact, in the various approaches logic programming is regarded prevalently as a formalism for knowledge representation, so that the semantics of a logic program is determined more by its "commonsense meaning" than by its purely logical contents. Completion or other transformations of the program are not needed any more, as it is interpreted directly in a non-monotonic framework.

In the rest of this chapter we will briefly describe the main approaches to the semantics of general programs which have been developed respectively within the completion-based and the declaratively-driven line of research. As an original result, we propose an extension of SLD-resolution based on the treatment of negative information via the NAI rule, the rule for negation introduced in Chapter 3. Furthermore, by a smooth amalgamation of the two approaches to negation, NAF and NAI, we formalize a system which combines the advantages of both of them, and allows us to deal with a class of programs larger than the class of general programs, namely that of programs whose clauses have any first-order formula as body (extended programs).

Such proposals fits in the first line of research, since they refer to the semantics of the extended completion, $Comp_L(P)$, defined in Section 3.2.4.

6.1 Completion-based approaches

After Clark's discovery ([23]) that negation as failure is sound with respect to the completed database (or completion), $Comp(P)$, of a (general) logic program P, it is becoming increasingly common to regard this as giving the appropriate semantics for logic programming. This explains why "negation as failure" is usually meant for SLDNF-resolution, i.e. the procedure resulting from adding negation as failure rule to SLD-resolution introduced in [23] (where it is called query evaluation procedure, QEP). The correctness of NAF w.r.t. $Comp(P)$ is expressed in [95] as follows (for the proof see [23] or [71]):

Let P be a general program and Q a general query. Then

1. if Q fails, then $Comp(P) \models \neg\exists Q$;
2. if Q succeeds with answer ϑ, then $Comp(P) \models Q\vartheta$,

where failure and success are referred to the application of SLDNF-resolution to $P \cup \{\leftarrow Q\}$.

However, concerning completeness we have to be aware that what Jaffar, Lassez and Lloyd ([56]) call the completeness of NAF is not at all a completeness result for SLDNF-resolution. In fact, since it has been shown only for positive programs and goals, the procedure is actually the ordinary SLD-resolution as the NAF rule is never invoked in the computation.

Indeed, all the works in this area are motivated by the desire to obtain completeness of (some extensions of) the SLDNF-resolution w.r.t. (some interpretations or extensions of) $Comp(P)$ for (some classes of) general programs and goals. It is worthwhile to point out that no completeness result is possible without restrictions, for essentially two reasons. One is the "classicity" of the completion theory which contrasts with the "intuitionistic" nature of the SLDNF-resolution, and causes inconsistency. Another problem rises from the very definition of SLDNF-resolution such as given by Clark ([23]). Under this definition, when a negative literal is selected the evaluation proceeds only if it is a ground literal $\neg A$, and if A succeeds then the evaluation of $\neg A$ fails; if A fails on every evaluation path, then $\neg A$ succeeds. This definition is needed to ensure a *safe* (definition later) use of the NAF rule, but completeness is now threatened by the possibility that the evaluation is held up because only negative non-ground literals occur in the goal (*floundering*).

If SLDNF-resolution and $Comp(P)$ are not modified, then the only chance to obtain completeness is to consider particular classes of general programs and goals whose syntactical form guarantees that none of the previous problems can occur. Clark already realized ([23]) that only "...with certain restrictions on the data base and its queries, query evaluation could fall short of a complete inference system", and singled out sufficient conditions under which this could be accomplished, by introducing the *hierarchical constraint* on the data base and the *allowedness* condition on the queries. These conditions has been extensively studied in [93, 94], where they are recognized to be very strong; for example the hierarchical condition forbids any recursive or mutually recursive definitions. In [15] it can be found an exhaustive list of the classes of general programs and goals for which SLDNF-resolution is complete and the related completeness results w.r.t. Comp(P). In that paper the ways to weaken the various conditions are also investigated, and some conditions based on dependencies between atoms, under which the completed program is consistent are defined.

The alternative is that of finding an appropriate expression for the reference theory by modifying (the notion of model of) $Comp(P)$, so that completeness can be achieved for SLDNF-resolution, or/and adopting more suitable forms of negation in the resolution procedure which avoids floundering.

6.1.1 Non-classical logics and completion

A disjunctive formula like $p \vee q$ can be proved by SLDNF-resolution (as in intuitionistic logic) only by proving p or q. On the contrary classical logic is provided with other means to prove such formulas, for example the excluded middle principle. Thus, it can happen that a formula is a logical consequence of $Comp(P)$, but the corresponding query Q is not answered by SLDNF-resolution which engages in an infinite computation with $P \cup \{\leftarrow Q\}$. In other words, completeness is not achieved. For example, for the program

$$P: \quad \{ \quad \begin{array}{l} p \leftarrow q \\ p \leftarrow \neg q \\ q \leftarrow q \quad \} \end{array}$$

p is a logical consequence of $Comp(P)$, since $p \leftrightarrow q \vee \neg q$ is a formula of $Comp(P)$ and $q \vee \neg q$ is classically valid. However, SLDNF-resolution will fail to halt with any query involving q, thus the query $\leftarrow p$ does not succeeds. The natural suggestion is then to consider the intuitionistic logical consequences of $Comp(P)$ instead of the classical ones. This would solve the problem with the previous example since $q \vee \neg q$ is not intuitionistically valid (and then p is not an intuitionistic logical consequence of $Comp(P)$). However, completeness is not yet obtained, since for the program

$$P_1: \ \{p \leftarrow \neg p\},$$

$Comp(P_1)$ is inconsistent both in classical and in intuitionistic logic. It is interesting noting that what actually causes incompleteness is non-termination of the interpreter combined with the inability of classical or intuitionistic logic to model such an eventuality. Actually, the best way to represent computations should be 3-valued logic, whose truth values, *true*, *false* and u (undefined), provide the logical counterparts of respectively success, failure and loop, i.e. the three possibilities of a computation. On the other hand, this logic was introduced by Kleene just to deal with partial recursive functions by disposing also of a mean for the description of computations which failed to return an answer.

In [66] and [67] Kunen uses a variant of Kleene's 3-valued logic to define the models of $Comp(P)$ to be considered in the semantics of the program P. In his approach the logical connectives $\neg, \vee$ and $\wedge$ are interpreted as in the truth tables given by Kleene ([63]), whereas the equivalence $\leftrightarrow$ is interpreted by an ad hoc three-valued operator $\cong$ which is not definable in terms of $\neg, \vee$ and $\wedge$ (the so-called Lukasiewicz's equivalence). The truth table of $A \cong B$ outputs the value *true* when both A and B are assigned the same truth values and *false* otherwise.

Thus, $p \cong p$ is indeed a tautology but $p \cong \neg p$ is not always false and the completion of the program P_1 above mentioned is not inconsistent anymore.

Kunen's semantics is usually referred to as the *3-valued completion* semantics. It is strongly related to the Fitting's operator Φ_P ([41]) which acts on partial (or 3-valued) interpretations. Its use allows to avoid the usual difficulty with general programs that the T_P operator is no longer monotonic. In fact, Φ_P is monotonic for all program P, even if continuity is not granted. The minimal 3-valued model of $Comp(P)$ is its least fixpoint $\Phi_P \uparrow \alpha$, and Kunen semantics is also defined in terms of Φ_P by the relation

$$\{L \mid \Phi_P \uparrow n \models_3 L,\ n < \omega\} = \{L \mid Comp(P) \models_3 L\},$$

where L is a literal and $\models_3$ indicates the logical consequence relation w.r.t. the 3-valued models. The completeness of SLDNF-resolution holds for all general programs and goals w.r.t. the 3-valued completion, under the hypothesis of non-floundering (Theorem 4.3 [67]).

It is important to point out that Kunen result is formulated in terms of ground substitutions, i.e. for the propositional case; but it can be invalidated if also quantifiers are taken into account, as Kunen showed by the following example:

Example 6.1.1

$$P: \quad \left\{ \begin{array}{l} p(x) \leftarrow isc(x) \\ p(x) \leftarrow nonc(x) \\ isc(c) \leftarrow \\ nonc(x) \leftarrow \neg isc(x) \quad \}. \end{array} \right.$$

SLDNF-resolution returns $x = c$ as an answer for the query $p(x)$, even if $\forall x.p(x)$ is a logical consequence of the 3-valued completion of P ($p(x) \cong x = c \vee x \neq c$ is a formula of the 3-valued completion of P), so that the empty answer substitution would be the correct one. The problem is that we need to construct "complete answers" for non-ground queries as intermediate steps. These complete answers need not to be just equations of the form $x = t$, but possibly negations or Boolean combination of these equations, and they should be obtained by amalgamating the answers on the single branches of the derivation tree. This is not possible for the linearity of SLDNF-resolution. Since $x = c \vee x \neq c$ is not intuitionistically valid, Shepherdson ([95]) suggested the possibility to obtain a stronger result by replacing the 3-valued logical consequence relation ($\models_3$) by some 'intuitionistic 3-valued derivability relation' ($\vdash_{3I}$). Afterwards, this idea has been realized in [108], where a 3-valued intuitionistic version of Clark's completion is presented, w.r.t. which completeness for SLDNF-resolution is proved, under the hypothesis of non-floundering.

Another proposal is that of [16], where linear logic is used. This choice is motivated by the absence in that logic of the contraction rule (a rule of the Gentzen formalization of classical logic known as sequent calculus), which seems to be responsible of the difficulties above mentioned, in obtaining completeness of SLDNF-resolution w.r.t. $Comp(P)$ (inconsistency of $Comp(P)$, combination of middle excluded principle and infinite computations). Clark's completion is reformulated by using linear connectives rather than the classical ones. The resulting semantics is the *linear completion* $LComp(W)$, w.r.t. which SLDNF-resolution is shown to be both sound and complete, under the hypothesis of non-floundering.

The last two results are both implied by Kunen's result; thus, Kunen's semantics can be thought of as the representative of the approaches based on non-classical logics.

A different approach is taken in [34], where intuitionistic logic is not explicitly considered, but a modified form of program completion is defined to take into account the behavior of the interpreter which, like intuitionistic logic, can prove $p \vee \neg p$ only if at least one of the elements of the disjunction can be really proved. This is obtained, roughly speaking, by introducing new predicate symbols to occur in negated literals of the clause bodies instead of the original ones in such a way that a clause $p \leftarrow \neg q$ is replaced by $p \leftarrow \neg q'$, and by adding copies of programs clauses for the new predicate symbols. The idea is to simulate the fact that when the interpreter tries to prove a negative ground literal it is not necessarily dealing with the predicate symbols of the original program but only with an exact copy of it. The transformed program is then completed in the usual way by obtaining the *strict completion* $comp^*(P)$. SLDNF-resolution is proved complete w.r.t. the new notion of completion under the hypothesis of non-floundering.

6.1.2 Floundering and constructive negation

All the completeness results mentioned in previous section hold *under the hypothesis of non-floundering*. It is then worthwhile to recall the reason why this problem does exist. It results from the use of *safe* (in the sense of [71]) selection rules in the SLDNF-resolution, imposing a negative literal to be selected only if it is ground. Otherwise, the procedure should not be correctly applied, since the query $\neg p(x)$, whose intended meaning is the formula $\exists x. \neg p(x)$, would be treated as if its meaning were $\forall x. \neg p(x)$. On the other hand, completeness of the proof procedure is "disturbed" since this might *flounder* with goals where all the literals are negative non-ground literals; in fact, in this case computation must block since no literal can be selected to go on. A more liberal version of NAF (see for instance § 15 in [71]) allows to reduce the possibility of floundering: $\leftarrow \neg A$ succeeds if $\leftarrow A$ has a failed SLD-tree, and it fails if $\leftarrow A$ has a refutation which does not instantiate the variables of A. However, the computation still arrests if none of these possibilities occur. In particular, it happens when $\leftarrow A$ has a refutation which instantiates the variables of A.

Apart from imposing syntactical restrictions on the programs and the goals so to ensure that floundering will never occur, a more radical way to tackle this problem is trying to appropriately deal with non-ground negative literals, rather than avoiding completely their treatment, thereby holding in due consideration also negative subgoals. In Section 5.1 we have mentioned the various proposals which have been suggested in order to transforming NAF into a more constructive negation rule. The effect of the introduction of constructive negation into the resolution procedure for dealing with general programs and goals (actually the extension of SLD-resolution with some constructive versions of NAF) is not immediate at all, due to the already mentioned complexity of the computed answer substitutions, even for very simple programs, when the negation rule is used nestedly (see Example 6.1.1).

The extension to general programs of the *fail-substitutions* approach proposed in [73] for positive programs is not quite straightforward; it has been appropriately defined in [35], where SLDFA-resolution (SLD-resolution with fail answers) is presented as an extension of SLDNF-resolution able to deriving answers for negative queries (fail answers). The problem of floundering is then removed, and then completeness of SLDFA-resolution is proved w.r.t. Kunen's semantics. The method used in this approach is a "syntactical method": only syntactical notions such as finitely failed tree and constraint[1] are exploited in order to construct a fail answer. This contrasts with all the other approaches which heavily use the formulas of the completion, and are, in this sense, tied to the completion semantics. They can be viewed, after all, as mechanisms to implement "computing in the completion". On the contrary, the independence from the completion makes Drabent's approach amenable also with other semantics. Actually a slight modification of SLDFA is shown correct w.r.t. the well-founded semantics.

[1] In this approach the term *constraint* is used to indicate an arbitrary first-order formula built out of equalities and inequalities.

Chan's constructive negation has been firstly formalized by the author in the SLD-CNF-resolution ([17]) where a negative literal $\neg A$ is solved, roughly, by constructing the complete SLD-tree for A and taking as the answer to $\neg A$ the negation of the answer substitutions (disjunction of equations) computed by the success paths in the tree. However, this negation rule cannot be applied when the SLD-tree is infinite or there are infinite answer substitutions for the positive atom. This problem has been solved in a later paper ([18]), where, instead of the complete SLD-tree, only a finite part of the SLD-tree is built, of which one cross-section chosen. However, for this schema no completeness result has been shown.

The technique of constructive negation proposed by Chan in the framework of logic programming has been afterwards generalized to clp scheme by Stuckey ([102]). The main result of that paper is the completeness of the operational mechanism defining the new constructive negation w.r.t. Kunen's 3-valued semantics.

The other technique of constructive negation, i.e. the compilative technique originally proposed in the context of positive programs by the transformational approach ([10]), has been recently generalized in [14] where it is used to define intensional negation for general programs in the framework of clp. A result of completeness is also shown for this negation w.r.t. the 3-valued completion semantics by using a semantic characterization similar to that of [102].

The extension of NAF proposed in [96] is, in a sense, "less constructive" than the above described negation rules: a negative literal $\neg A$ is selected in a goal if *there exists* a substitution ϑ such that $A\vartheta$ fails and, in this case, ϑ is applied to the resultant goal. Thus, one has to consider all possible substitutions to apply to the negative literals, since no ways to construct them are specified. In the resulting procedure, called SLDNFS-resolution, non-ground negative literals are not allowed to be selected in the construction of the auxiliary finitely failed trees, but only in the main derivation. Anyway, floundering is excluded and a completeness result is proved. Differently from the other approaches, where the existent completeness results have been all shown w.r.t. Kunen's semantics, in [96] a weak version of the completion semantics is defined, which is based on an iterative construction, and SLDNFS-resolution is shown to be correct and complete w.r.t. it.

6.1.3 Proof-theory of general programs

The relationship between proof-theory and logic programming has been pointed out be Girard ([48]) by using the Gentzen sequent calculus: for definite programs, SLD-computations can be transformed into sequent proofs and vice-versa. In recent works [99, 100] these correspondence has been studied also for general programs and SLDNF-resolution. In order to formalize in a sequent calculus computations invoking the NAF rule, the completion of the program is used to define some non-logical axioms and non-logical rules of the calculus. In the resulting system, every computation has a corresponding sequent proof. However, the converse is not possible in general, but only for a particular class of programs (called *decomposable* programs) which contains definite and allowed programs. The sequent calculus is shown to be a sound and complete base for a deductive approach to SLDNF-resolution. In fact, completeness of SLDNF-resolution is shown for decomposable programs w.r.t. the Kunen's semantics.

Even tough the approach is completely atypical for the tradition of the semantics of logic programming, we can classify it as a completion-based approach since the basic idea is still to think of the meaning of a logic program as determined by its completion.

6.2 The SLDNI-resolution

Of the two aspects characterizing constructive negation, namely the ability of computing bindings for the variables in the negative subgoals, and the elimination of the floundering in the systems based on it, NAI rule retains only the latter. In fact, it is, like NAF, only a test for negative literals (thus it does not returns bindings), and differently from NAF it is able to deal with non-ground (existentially closed) negative literals (thus floundering is excluded). In this section we consider the application of the NAI rule to general programs. In order to stress the specialization of this rule for the treatment of existentially closed negative queries we introduce explicitly the existential quantifier in the syntax by considering programs which might have existentially quantified negative queries in the bodies of the clauses. We call them $\exists\neg$*general* programs. Then, we extend SLD-resolution by adding NAI rule to solve these subgoals, thus obtaining what we call SLDNI-resolution.

The syntax for $\exists\neg$*general* queries, goals and programs is described by the following grammar:

Queries	$Q ::= \ A \mid \exists\neg(Q) \mid Q, Q$
Goals	$G ::= \leftarrow Q$
Clauses	$C ::= \ A \leftarrow Q \mid A \leftarrow .$

where $A \in Atom$.

In the rest of the paper, when there is no risk of ambiguity, we write $\exists\neg Q$ for $\exists\neg(Q)$.

The notions of literal and selection rule are extended as follows.

Definition 6.2.1 *Let Q be a query. The* conjuncts *of Q, Conj(Q), is the smallest set such that*

- *if $Q \equiv A$ then Conj(Q)=$\{A\}$*
- *if $Q \equiv \exists\neg Q'$ then $Conj(Q) = \{\exists\neg Q'\}$*
- *if $Q \equiv Q_1, Q_2$ then $Conj(Q) = Conj(Q_1) \cup Conj(Q_2)$.*

Definition 6.2.2 *A* selection rule *specifies, for every goal $\leftarrow Q$, one conjunct of Q (the selected conjunct).*

Roughly, the notion of derivation extends as follows. Let P be a $\exists\neg$*general* program, R a selection rule and $\leftarrow Q$ the current goal. Let Q' be the selected conjunct. If Q' is an atom, then we choose a clause $A \leftarrow Q''$ of P_{ren} such that A unifies with Q', and replace Q' by Q'', in Q. Finally, we apply the *mgu* θ to all variables which are not in the scope of an existential quantifier. We will use the notation $\leftarrow ((Q \setminus Q'), Q'')\theta$ to indicate the resulting goal. If such a clause does not exist, then the derivation fails.

If Q' is of the form $\exists\neg Q''$, then we check whether the derivation tree for $P \cup \{\leftarrow Q''\}$ instantiates the free variables of Q''. If this is the case, then Q' is removed from $\leftarrow Q$. The resulting goal will be indicated with $\leftarrow Q \setminus Q'$. If there exists a refutation for $P \cup \{\leftarrow Q''\}$ which does not instantiate the free variables of Q'', then the derivation fails.

The derivation succeeds if it ends with the empty goal.

Note that there are derivations which neither succeed nor fail. They can be infinite, like in $\{p \leftarrow p\} \cup \{\leftarrow p\}$, or loop forever in the attempt to solve a negative conjunct, like in $\{p \leftarrow p\} \cup \{\leftarrow \exists\neg p\}$ and in $\{p \leftarrow \exists\neg p\} \cup \{\leftarrow p\}$.

In the following, we denote by $freevar(Q)$ the free variables in Q, i.e.

- if $Q \equiv A$, then $freevar(Q) = var(A)$,
- if $Q \equiv \exists\neg Q''$, then $freevar(Q) = \emptyset$,
- if $Q \equiv Q_1, Q_2$, then $freevar(Q) = freevar(Q_1) \cup freevar(Q_2)$.

Following Shepherdson ([96]), we formalize the extended notion of refutation and tree. We call them *SLDNI-refutation* and *SLDNI-tree*, for SLD with Negation As Instantiation. First, let's make precise the notion of application of a substitution to a $\exists\neg$*general* query.

Definition 6.2.3 *Let Q be a $\exists\neg$general query and θ be a substitution. Then*

$$Q\theta = \begin{cases} A\theta & \text{if } Q = A \\ Q & \text{if } Q = \exists\neg Q' \\ Q_1\theta, Q_2\theta & \text{if } Q = Q_1, Q_2. \end{cases}$$

Analogously, the notation $Q[\bar{d}/\bar{x}]$ indicates the replacement of $\bar{d}$ for the free occurrences of $\bar{x}$ in Q.

Definition 6.2.4 (SLDNI-resolution) *Let P be a $\exists\neg$general program, R a selection rule and G a $\exists\neg$general goal.*

- *An* SLDNI-tree of rank k *for $P \cup \{G\}$, via R, is a tree TR defined as follows:*
  (a) *Every node is a $\exists\neg$general goal and every edge is labeled by a substitution,*
  (b) *the root node is G,*
  (c) *for every node $\leftarrow Q$*
    (i) *if the selected conjunct is an atom A, then for each clause $H \leftarrow Q'$ in P (standardized apart), if H and A are unifiable with mgu θ, then the goal $\leftarrow ((Q \setminus A), Q')\theta$ is a child of $\leftarrow Q$, the edge is labeled by θ and the conjunct is said to succeed. If there are no such children the node is a leaf node and the conjunct is said to fail;*
    (ii) *if the selected conjunct is $\exists\neg Q'$ and there exists a finitely instantiating SLDNI-tree for $P \cup \{\leftarrow Q'\}$ of rank $k' < k$, then $\leftarrow Q \setminus \exists\neg Q'$ is the only child of $\leftarrow Q$, the edge is labeled by ε, and the conjunct is said to succeed. If there exists a non-instantiating SLDNI-refutation of rank $k' < k$ for $P \cup \{\leftarrow Q'\}$, then $\leftarrow Q$ is a leaf node, and the conjunct is said to fail.*
- *TR is* finitely instantiating *at level h iff it instantiates $freevar(Q)$ at level h (see Definition 3.2.2).*
- *An* SLDNI-refutation ξ of rank k *for $P \cup \{G\}$ is a sequence of goals $\xi = G_0, G_1, \ldots, G_n$, with $G_0 = G$ and $G_n = \Box$, and an associated sequence of substitutions $\theta_0, \ldots, \theta_{n-1}$ such that for each $i \in [0, n-1]$, either*
  (i) *the selected conjunct in G_i is an atom A, there exists a clause $H \leftarrow Q$ in P (standardized apart) such that H is unifiable with A with mgu θ_i and $G_{i+1} = \leftarrow ((G_i \setminus A), Q)\theta_i$, or*

(*ii*) *the selected conjunct in* G_i *is* $\exists\neg Q$*, there exists a finitely instantiating SLDNI-tree of rank* $k' < k$ *for* $P \cup \{\leftarrow Q\}$*,* $G_{i+1} = G_i \setminus \exists\neg Q$ *and* $\theta_i = \varepsilon$*.*
The substitution $(\theta_0 \cdots \theta_{n-1})_{|G}$ *is the* computed answer substitution.

- ξ *is* non-instantiating *if* $Inst(\theta_0 \cdots \theta_{n-1}, freevar(G))$ *is false.*

Note that a particular case of finitely instantiating SLDNI-tree is a tree in which all branches fail, which we call finitely failed SLDNI-tree.

6.2.1 Correctness of SLDNI-resolution

In this section we define the completion of a $\exists\neg$*general* program and we show that the SLDNI-resolution is sound with respect to it, thus extending the results of Section 3.2.1 to $\exists\neg$*general* programs.

The completion of a $\exists\neg$*general* program is just the straightforward extension of the standard notion of completion recalled in Definition 3.2.18 to $\exists\neg$*general* programs. However we need here to give more details about its definition, because we are going to use it in the proofs.

Definition 6.2.5 *Let* P *be a* $\exists\neg$*general program. Its completion with respect to the extended language* L *is defined in the usual way, namely* $Comp_L(P) = IFF(P) \cup CET_L$*. The set* CET_L *is the set of the Clark's equality axioms for the language* L *([23]). The set* $IFF(P)$ *is the collection of completed definitions of predicates in* P*, defined as follows.*
Let p *be a predicate occurring in the program and* $x_1, \ldots, x_n$ *fresh variables. Assume there are* $s \geq 0$ *clauses in* P *defining* p *(i.e. with head predicate* p*). For* $i \in [1, s]$*, let* $p(\bar{t_i}) \leftarrow Q_i$ *be the* i*-th clause in* P *defining* p*. Let* E_i *be the formula* $\exists \bar{y_i}.\bar{x} = \bar{t_i} \wedge Q_i$*, where* $\bar{y_i}$ *denotes all the (free) variables in* $p(\bar{t_i}) \leftarrow Q_i$*. The completed definition of* p *in* P *is*

$$p(\bar{x}) \leftrightarrow \bigvee_{i \in [1,s]} E_i.$$

We remind that the symbol "," in the queries has to be interpreted as the logical conjunction "$\wedge$". In the following, when the program P is clear from the context, we write $\models F$ for $Comp_L(P) \models F$.

Proposition 6.2.6 *Let* P *be a* $\exists\neg$*general program, and* $\leftarrow Q$ *a* $\exists\neg$*general goal. Consider an SLDNI-tree for* $P \cup \{\leftarrow Q\}$*. Assume the selected conjunct in* $\leftarrow Q$ *to be an atom* A*. Let* $\bar{x} = x_1, \ldots, x_m \in freevar(Q)$ $(m \geq 0)$*.*

1. *If* $\leftarrow Q'$ *is a child of* $\leftarrow Q$*, with associated substitution* θ*, then*

$$Comp_L(P) \models Q\theta \leftarrow Q'.$$

2. *If* $\leftarrow Q_1, \ldots, \leftarrow Q_n$ $(n \geq 0)$ *are all the children of* $\leftarrow Q$ *whose associated substitutions* $\theta_1, \ldots, \theta_n$ *satisfy* $\forall i \in [1, n]$ $\neg Inst(\theta_i, \bar{x})$*, then*

$$Comp_L(P) \models \neg Q[\bar{d}/\bar{x}] \leftarrow \bigwedge_{i \in [1,n]} \forall \bar{y_i}.\neg Q_i[\bar{d}/\bar{x}\theta_i],$$

where $\bar{d} = d_1, \ldots, d_m$ *are fresh constant symbols and the* $\bar{y_i}$*'s are all the local variables of the clauses used in the derivation steps, after renaming.*

Proof

1. This part extends to $\exists\neg$*general* programs the result stated (without proof) in [4], Lemma 3.1.

   Let $H \leftarrow Q''$ be the selected clause. Then

$$
\begin{array}{ll}
 & \models H \leftarrow Q'' \\
\Rightarrow & \\
 & \models H\theta \leftarrow Q''\theta \\
\Rightarrow & \quad \{\text{since } A\theta = H\theta\} \\
 & \models A\theta \leftarrow Q''\theta \\
\Rightarrow & \\
 & \models (Q \setminus A)\theta, A\theta \leftarrow (Q \setminus A)\theta, Q''\theta \\
\Rightarrow & \quad \{\text{since } Q' = ((Q \setminus A), Q'')\theta = (Q \setminus A)\theta, Q''\theta\} \\
 & \models Q\theta \leftarrow Q'.
\end{array}
$$

2. Assume $A = p(\bar{t})$. Let $p(\bar{y}) \leftrightarrow \bigvee_{j\in[1,s]} \exists\bar{z_j}.\bar{y} = \bar{u_j} \wedge Q'_j$ be the completed definition of p, where all variables are renamed according to the standardization apart. Then

$$
\begin{array}{ll}
 & \models A \leftrightarrow \bigvee_{j\in[1,s]} \exists\bar{z_j}.\bar{t} = \bar{u_j} \wedge Q'_j \\
\Rightarrow & \\
 & \models A[\bar{d}/\bar{x}] \leftrightarrow \bigvee_{j\in[1,s]} \exists\bar{z_j}.\bar{t}[\bar{d}/\bar{x}] = \bar{u_j} \wedge Q'_j.
\end{array}
$$

   Observe that, for $j \in [1, s]$, if $\bar{t}$ and $\bar{u_j}$ are not unifiable, then the component $F_j = \exists\bar{z_j}.(\bar{t}[\bar{d}/\bar{x}] = \bar{u_j} \wedge Q'_j)$ is false in $Comp_L(P)$, hence we can discard it from the disjunction in the right hand side.

   Analogously, if there exists $\mu_j = mgu(\bar{t}, \bar{u_j})$, but $Inst(\mu_j, \bar{x})$ holds, then $\bar{t}[\bar{d}/\bar{x}]$ and $\bar{u_j}$ are not unifiable, hence we can discard the component F_j.

   Finally, if $\mu_j = mgu(\bar{t}, \bar{u_j})$ and $\neg Inst(\mu_j, \bar{x})$, then $mgu(\bar{t}[\bar{d}/\bar{x}], \bar{u_j}) = \mu_j[\bar{d}/\bar{x}\mu_j]$, hence

$$\models F_j \rightarrow \exists\bar{z_i}.Q'_j\mu_j[\bar{d}/\bar{x}\mu_j].$$

   Let $j_1, \ldots, j_n \in [1, s]$ be the indexes such that $\exists\ mgu(\bar{t}, \bar{u_{j_k}}) = \mu_{j_k}$ and $\neg Inst(\mu_{j_k}, \bar{x})$. We have

$$
\begin{array}{ll}
 & \models A[\bar{d}/\bar{x}] \rightarrow \bigvee_{k\in[1,n]} \exists\bar{z_i}.Q'_{j_k}\mu_{j_k}[\bar{d}/\bar{x}\mu_{j_k}] \\
\Rightarrow & \quad \{\text{since } Q = ((Q \setminus A), A)\} \\
 & \models Q[\bar{d}/\bar{x}] \rightarrow \bigvee_{k\in[1,n]} \exists\bar{z_i}.(Q \setminus A)[\bar{d}/\bar{x}\mu_{j_k}], Q'_{j_k}\mu_{j_k}[\bar{d}/\bar{x}\mu_{j_k}]) \\
\Leftrightarrow & \quad \{\text{definition of the } Q_i\text{'s}, \bar{y_i}\text{'s, and } \theta_i\text{'s}\} \\
 & \models Q[\bar{d}/\bar{x}] \rightarrow \bigvee_{i\in[1,n]} \exists\bar{y_i}.Q_i[\bar{d}/\bar{x}\theta_i] \\
\Leftrightarrow & \\
 & \models \neg Q[\bar{d}/\bar{x}] \leftarrow \bigwedge_{i\in[1,n]} \forall\bar{y_i}.\neg Q_i[\bar{d}/\bar{x}\theta_i].
\end{array}
$$

□

Note that for $m = 0$ (i.e. $\bar{x} = \emptyset$) Proposition 6.2.6(2) corresponds to Lemma 3.15 of [71], and for $n = 0$ (i.e. when all resolvents instantiate $\bar{x}$) it reduces to $Comp_L(P) \models \neg Q[\bar{d}/\bar{x}]$.

Theorem 6.2.7 (Correctness of SLDNI-resolution) *Let P be a $\exists\neg$general program and $\leftarrow Q$ a $\exists\neg$general goal. Then*

1. *if $P \cup \{\leftarrow Q\}$ has an SLDNI-refutation with c.a.s. θ, then $Comp_L(P) \models Q\theta$,*

2. *if $P \cup \{\leftarrow Q\}$ has a finitely instantiating SLDNI-tree, then $Comp_L(P) \models \exists\neg Q$.*

Proof By mutual generalized induction on the rank k.

1. For this part we reason by induction w.r.t. both the rank k and the length n of the refutation. We use the lexicographic ordering on pairs $\langle k, n\rangle$ of natural numbers. In this ordering

   $$\langle k_1, n_1\rangle < \langle k_2, n_2\rangle \text{ iff } k_1 < k_2 \text{ or } (k_1 = k_2 \text{ and } n_1 < n_2).$$

   If $n = 0$, then $\leftarrow Q$ is the empty goal.

   Assume now $n > 0$. We distinguish the two cases.

   (i) The selected conjunct is an atom. Let μ be the mgu and $\leftarrow Q'$ the resolvent of $\leftarrow Q$. Then $P \cup \{\leftarrow Q'\}$ has an SLDNI-refutation of length $n-1$ with a c.a.s. θ' such that $Q\mu\theta' = Q\theta$. By inductive hypothesis, $\models Q'\theta'$ holds. Hence, by Proposition 6.2.6(1), we have $\models Q\mu\theta'$, i.e. $\models Q\theta$.

   (ii) The selected conjunct is a negative query $\exists\neg Q'$, and $P \cup \{\leftarrow Q'\}$ has a finitely instantiating SLDNI-tree of rank $k' < k$. Furthermore, $P \cup \{\leftarrow (Q \setminus \exists\neg Q')\}$ has an SLDNI-refutation of length $n-1$ with a c.a.s. θ. Then, by inductive hypothesis, we have $\models \exists\neg Q'$ and $\models (Q \setminus \exists\neg Q')\theta$. Finally observe that $((Q \setminus \exists\neg Q')\theta, \exists\neg Q') = Q\theta$ holds, since, by Definition 6.2.3, $(\exists\neg Q')\theta = \exists\neg Q'$ and thus we have $\models Q\vartheta$.

2. For this part we reason by induction w.r.t. both the rank k and the instantiation level h. We use the lexicographic ordering on pairs $\langle k, h\rangle$ of natural numbers. Note that changing the ordering w.r.t. Part 1 is harmless, since the mutual inductive hypothesis is done on k strictly smaller, and k is the first component of the pairs in both the lexicographyc orderings.

   Assume that $P \cup \{\leftarrow Q\}$ has an SLDNI-tree which finitely instantiates at level h some variables $\bar{x} = x_1, \ldots, x_m \in \mathit{freevar}(Q)$. We prove a stronger property, namely that $Comp_L(P) \models \neg Q[\bar{d}/\bar{x}]$ holds. Consider the two cases:

   (i) The selected conjunct is an atom.
   Consider all the children $\leftarrow Q_1, \ldots, \leftarrow Q_n$, $(n \geq 0)$, of $\leftarrow Q$, whose associated substitutions $\theta_1, \ldots, \theta_n$ satisfy $\neg Inst(\theta_i, \bar{x})$. By definition, for each $i \in [1, n]$, $P \cup \{\leftarrow Q_i\}$ has an SLDNI-tree which finitely instantiates $\bar{x}\theta_i$ at level $h_i < h$ (note that $\bar{x}\theta_i$ is a sequence of variables). By inductive hypothesis, $\forall i \in [1, n]$, $\models \neg Q_i[\bar{d}/\bar{x}\theta_i]$, hence $\models \bigwedge_{i \in [1,n]} \forall \bar{y}_i . \neg Q_i[\bar{d}/\bar{x}\theta_i]$ holds, where the $\bar{y}_i$ are the local variables of the clauses used in the derivation steps, after renaming. Finally, apply Proposition 6.2.6(2).

   (ii) The selected conjunct is a negative query $\exists\neg Q'$. Consider the two cases:

   (a) $P \cup \{\leftarrow Q'\}$ has an SLDNI-refutation of rank $k' < k$ with a c.a.s. θ such that $\neg Inst(\theta, \mathit{freevar}(Q'))$, namely there exists θ' such that $Q'\theta\theta' = Q'$. By inductive hypothesis, $\models Q'\theta$ holds, hence we deduce $\models Q'\theta\theta'$, i.e. $\models Q'$. By simple formula manipulation we have:

$$\begin{array}{l} \quad \models Q' \\ \Leftrightarrow \\ \quad \models \neg\exists\neg Q' \\ \Rightarrow \\ \quad \models \neg(Q \setminus \exists\neg Q')[\bar{d}/\bar{x}] \vee \neg\exists\neg Q' \\ \Rightarrow \\ \quad \models \neg((Q \setminus \exists\neg Q')[\bar{d}/\bar{x}], \exists\neg Q'). \end{array}$$

Finally, observe that $(Q \setminus \exists\neg Q')[\bar{d}/\bar{x}], \exists\neg Q' = Q[\bar{d}/\bar{x}]$.

(*b*) $P \cup \{\leftarrow Q'\}$ has a finitely instantiating SLDNI-tree of rank $k' < k$, so the (only) child of $\leftarrow Q$ is $\leftarrow (Q \setminus \exists\neg Q')$. By hypothesis, $P \cup \{\leftarrow (Q \setminus \exists\neg Q')\}$ has a finitely instantiating SLDNI-tree of rank $k'' < k$. By inductive hypothesis we have $\models \neg(Q \setminus \exists\neg Q')[\bar{d}/\bar{x}]$. By simple formula manipulation:

$$\begin{array}{l} \quad \models \neg(Q \setminus \exists\neg Q')[\bar{d}/\bar{x}] \\ \Rightarrow \\ \quad \models \neg(Q \setminus \exists\neg Q')[\bar{d}/\bar{x}] \vee \neg\exists\neg Q' \\ \Rightarrow \\ \quad \models \neg((Q \setminus \exists\neg Q')[\bar{d}/\bar{x}], \exists\neg Q'). \end{array}$$

Finally, use again the equality $(Q \setminus \exists\neg Q')[\bar{d}/\bar{x}], \exists\neg Q' = Q[\bar{d}/\bar{x}]$.

□

6.2.2 Incompleteness of SLDNI-resolution

The SLDNI-resolution excludes floundering, but it suffers from a problem similar to the problem of SLDNF-resolution which in Section completion-based we have ascribed to the classicity of the completion semantics. The last problem is related to the infinite computations and corresponds to what Shepherdson calls a *dead end* ([94]). In case there is an infinite fair computation for $\leftarrow A$, then the mainstream evaluation of $\leftarrow \neg A$ is stuck because no answer is received about the subsidiary evaluation.

Example 6.2.8

$$\begin{array}{lcl} P & = & \{ \; p \leftarrow \exists\neg q(x), \\ & & \quad\; p \leftarrow q(x), \\ & & \quad\; q(x) \leftarrow q(x) \quad \} \end{array}$$

As one can easily verify, $Comp_L(P) \models p$ holds, but $P \cup \{\leftarrow p\}$ has no refutations. This is due to a dead end: neither SLDNI-refutations, nor finitely instantiating SLDNI-trees can be constructed for the subgoal $\leftarrow q(x)$.

Of course, if the program and the query are positive (i.e. they do not contain existential negative conjuncts), the dead end problem cannot occur because the NAI rule is never invoked during the computation. So, in this case we have for the NAI-rule a completeness result (see Proposition 3.2.22 and its extension to the non-atomic goals) which, paraphrasing Shepherdson's terminology, we call $\exists\neg$*completeness* of SLDNI-resolution.

Observe that $\exists\neg$*completeness* is a further way to intend completeness in addition to the ones considered by Shepherdson in [95]. Shepherdson distinguishes three kinds of completeness for SLDNF-resolution:

(θ-completeness) If $Comp(P) \models Q\theta$, then $P \cup \{\leftarrow Q\}$ succeeds with a c.a.s. θ' s.t. $\theta' \leq \theta$;

($\neg$-completeness) If $Comp(P) \models \neg\exists Q$, then $P \cup \{\leftarrow Q\}$ fails ;

($\exists$-completeness) If $Comp(P) \models \exists Q$ then $P \cup \{\leftarrow Q\}$ succeeds .

We can consider θ-completeness dual of $\neg$-completeness both from a logical, and from an operational point of view.

The $\exists\neg$*completeness* can be thought of as the dual of $\exists$-completeness and stated as follows:

($\exists\neg$ completeness) If $Comp_L(P) \models \exists\neg Q$ then $P \cup \{\leftarrow Q\}$ instantiates *freevar*(Q).

6.3 Amalgamating NAF and NAI

A major limitation of the language presented in Section 6.2 is that the negative components of a query cannot share variables with the positive ones. The situation is in a sense opposite to what happens in SLDNF-resolution, where, in order to avoid floundering, all the variables in the negative literals must occur in some positive atom too (in the clause or in the query). It comes then natural to try an amalgamation of NAF and NAI in order to reduce the limitations of the two methods. In the following we present two ways to do so, which we formalize respectively as SLDNFI-resolution ([32, 31]) and NAIF-resolution ([29]). The latter is in a sense an "abbreviation" of the former aimed at improving its efficiency, and it will be used for dealing with a more expressive language, namely the language of extended programs, already used in [72] in connection with SLDNF-resolution.

6.3.1 The SLDNFI-resolution

In order to introduce the possibility that some variables in a negative query could be shared with other queries, we must allow the presence of some free variables in the query. Thus we modify our language by allowing the presence of expressions like $\exists\bar{x}\neg(Q)$, where $\bar{x}$ is a (possibly empty) sequence of variables, and consider a slight alteration of the NAI rule so that $\leftarrow \exists\bar{x}\neg(Q)$ is evaluated successfully in P if there exists a tree which finitely instantiates $\bar{x}$. The NAF rule and the NAI rule are particular instances of this rule, obtained with $\bar{x} = \emptyset$ and $\bar{x} = \mathit{freevar}(Q)$ respectively.

Concerning the treatment of failure, it is still correct to infer the failure of $P \cup \{\leftarrow \exists\bar{x}\neg Q\}$ when there exists a refutation for $P \cup \{\leftarrow Q\}$ which does not instantiate the variables of Q. Note that, for correctness, it is necessary to consider all variables of Q, not only $\bar{x}$. This is similar to the NAF case.

Example 6.3.1 *Consider the program*

$$P = \{ \; p \leftarrow \exists\neg q, \quad q \leftarrow \exists x \neg r(x,y), \quad r(x,a) \leftarrow \; \}$$

Observe that $Comp_L(P) \not\models p$. Therefore a rule allowing to derive the falsity of q from the refutation of $\leftarrow \neg r(x,y)$ would be incorrect. Note that such a refutation instantiates y.

The arrest of the computation (with no conclusion) is limited to the situation in which there exists a refutation instantiating $\mathit{freevar}(Q) \setminus \bar{x}$. Note that our treatment of negation (for $\bar{x} = \emptyset$) is more general than the standard SLDNF-resolution as formalized, for instance, by Lloyd [71]. Usually SLDNF-resolution requires a selected negative literal to be ground, otherwise it flounders. We allow the selected conjunct to be not ground, and stop the computation (with no conclusion) only if there exists a refutation instantiating its free variables. By noting that for $\bar{x} = \emptyset$ a finitely instantiating tree is nothing else that a finitely failed tree, this treatment is rather similar to the liberal version of SDNF-resolution which we mentioned in Section 6.1.2.

The language amalgamating NAF and NAI is defined as follows:

Queries	$Q ::= \; A \mid \exists \bar{x} \neg (Q) \mid Q, Q$
Goals	$G ::= \leftarrow Q$
Clauses	$C ::= \; A \leftarrow Q \mid A \leftarrow .$

where $A \in \mathit{Atom}$ and $\bar{x} \in \mathit{Var}$. As usual, we omit the parenthesis when there is no risk of ambiguity.

Note that this language allows us to express various kinds of general formulas in the body of clauses. For instance, $\neg Q$ can be expressed as $\exists \emptyset \neg Q$; $\forall \bar{x} Q$ can be expressed as $\neg \exists \bar{x} \neg Q$; $\exists \bar{x}(Q_1 \rightarrow Q_2)$ can be expressed as $\exists \bar{x} \neg (Q_1, \neg Q_2)$; etc.

The notion of conjunct is modified according to the new grammar in the obvious way. So, a conjunct is either an atom or an expression of the form $\exists \bar{x} \neg Q$. As one would expect, the application of a substitution to a query generalizes as follows:

$$Q\theta = \begin{cases} A\theta & \text{if } Q = A \\ (\exists \bar{x} \neg Q')\theta_{|\mathit{Var} \setminus \bar{x}} & \text{if } Q = \exists \bar{x} \neg Q' \\ Q_1\theta, Q_2\theta & \text{if } Q = Q_1, Q_2. \end{cases}$$

From this definition it appears clear that quantified negative conjuncts can now communicate, i.e. share variables, with the other conjuncts.

We present now the resolution mechanism for the amalgamated language, which we call SLDNFI-resolution. This is obtained by modifying Definition 6.2.4 according to the above described treatment of formulas like $\exists \bar{x} \neg Q$.

Definition 6.3.2 (SLDNFI-resolution)

- *The notion of SLDNFI-tree is obtained by replacing the point c(ii) in Definition 6.2.4 by the following condition:*

  (ii) if the selected conjunct is $\exists \bar{x} \neg Q'$, then

  - *if there exists an SLDNFI-tree for $P \cup \{\leftarrow Q'\}$ of rank $k' < k$ which finitely instantiates $\bar{x}$, then $\leftarrow Q \setminus \exists \bar{x} \neg Q'$ is the only child of $\leftarrow Q$, and the edge is labeled by ε.*
  - *if there exists an SLDNFI-refutation for $P \cup \{\leftarrow Q'\}$ of rank $k' < k$ which does not instantiate $\mathit{freevar}(Q')$, then $\leftarrow Q$ is a (failed) leaf node.*
    (Note that the restriction concerning instantiation involves all the variables of Q', not only $\bar{x}$).

- *The notion of SLDNFI-refutation is obtained by replacing the corresponding point* (ii) *in Definition 6.2.4 by the condition:*

  (ii) *the selected conjunct in* G_i *is* $\exists\bar{x}\neg(Q)$, *there exists an SLDNFI-tree of rank* $k' < k$ *for* $P \cup \{\leftarrow Q\}$ *which finitely instantiates* $\bar{x}$, $G_{i+1} = G_i \setminus \exists\bar{x}\neg(Q)$ *and* $\theta_i = \varepsilon$.

The following result states the correctness of SLDNFI-resolution, thus generalizing Theorem 6.2.7.

Theorem 6.3.3 (Correctness of SLDNFI-resolution) *Let* P *be an amalgamated program,* $\leftarrow Q$ *an amalgamated goal and* $\bar{x} \in freevar(Q)$. *Then*

1. *if* $P \cup \{\leftarrow Q\}$ *has an SLDNFI-refutation with c.a.s.* θ, *then* $Comp_L(P) \models Q\theta$,
2. *if* $P \cup \{\leftarrow Q\}$ *has an SLDNFI-tree finitely instantiating* $\bar{x}$, *then* $Comp_L(P) \models \exists\bar{x}\neg Q$.

Proof Slight modification of the proof of Theorem 6.2.7. In $1(ii)$ replace $\exists\neg Q'$ by $\exists\bar{z}\neg Q'$, and observe that $\models \exists\bar{z}\neg Q' \Rightarrow \models (\exists\bar{z}\neg Q')\theta$, and that $(Q \setminus \exists\bar{z}\neg Q')\theta, (\exists\bar{z}\neg Q')\theta = Q\theta$.
In $2(ii)(a)$ and $2(ii)(b)$ replace $\exists\neg Q'$ by $\exists\bar{y}\neg Q'$, and observe that $(Q \setminus \exists\bar{y}\neg Q')[\bar{d}/\bar{y}], \exists\bar{y}\neg Q' = Q[\bar{d}/\bar{y}]$. Furthermore, in $2(ii)(a)$ observe that $\models Q' \Rightarrow \models \forall\bar{y}Q' \Leftrightarrow \models \neg\exists\bar{y}\neg Q'$.
The rest of the proof remains the same. □

As a particular case of Theorem 6.3.3, when $\bar{x} = \emptyset$ we obtain:

Remark 6.3.4 *If* $P \cup \{\leftarrow Q\}$ *has a finitely failed SLDNFI-tree, then* $Comp_L(P) \models \neg Q$, *i.e.* $Comp_L(P) \models \forall\neg Q$.

6.3.2 The NAIF-resolution

Remark 6.3.4 suggests us the possibility of extending the language defined in Section 6.3.1 by introducing a "direct" universal quantifier. In particular, queries of the form $\forall\bar{x}\neg Q$ can be expressed in this language only by complicated queries such as $\exists\emptyset\neg(\exists\bar{x}\neg(\exists\emptyset\neg(Q)))$, obtained by considering $\forall\bar{x}.Q$ and $\neg Q$ as abbreviations for $\neg(\exists\bar{x}\neg(Q))$ and $\exists\emptyset\neg(Q)$. Thus, for solving $\forall\bar{x}\neg Q$, we have to evaluate three nested negations before evaluating $\leftarrow Q$. On the contrary, by using Remark 6.3.4 we could, for instance, directly infer $\forall\bar{x}\neg(Q)$ from the finite failure of the tree for $\leftarrow Q$, thus improving the efficiency of the interpreter.

In the following we define a language which admits both existentially quantified negative queries $\exists\bar{x}\neg Q$, and universally quantified negative queries $\forall\bar{x}\neg Q$. We call this language, $\Gamma-$*general* language, Γ standing for $\exists$ or $\forall$.

The syntax for $\Gamma-$*general* queries, $\Gamma-$*general* goals and $\Gamma-$*general* programs is described in Table 6.1, where A is an atom.

The notions of conjunct and application of a substitution to a query trivially generalize to $\Gamma-$*general* language. The resolution mechanism is similar to the SLDNFI-resolution, but it is able to deal with universally quantified queries too. Roughly, when a conjunct of the form $\forall\bar{x}\neg(Q)$, is selected then we check whether the derivation tree for $P \cup \{\leftarrow Q\}$ is finitely failed. If this is the case, then the conjunct succeeds; if there exists a refutation for $P \cup \{\leftarrow Q\}$ which does not instantiate the free variables of $\forall\bar{x}\neg(Q)$, then the conjunct fails.

We call the resulting interpreter NAIF-resolution for SLD-resolution with Negation As Instantiation and Failure. It is defined by extending Definition 6.3.2 according to the above described treatment of the formulas $\forall\bar{x}\neg(Q)$.

Definition 6.3.5 (NAIF-resolution)

Table 6.1: The Γ–*general* language

Queries	$Q ::= A \mid \exists\bar{x}\neg(Q) \mid \forall\bar{x}\neg(Q) \mid Q, Q$
Goals	$G ::= \leftarrow Q$
Clauses	$C ::= A \leftarrow Q \mid A \leftarrow .$

- *The notion of NAIF-tree is obtained by adding the following condition to the Definition 6.3.2:*

  (iii) if the selected conjunct is $\forall\bar{x}\neg Q'$, then

  - *if there exists a finitely failed NAIF-tree for $P \cup \{\leftarrow Q'\}$ of rank $k' < k$, then $\leftarrow Q \setminus \forall\bar{x}\neg(Q')$ is the only child of $\leftarrow Q$, and the edge is labeled by ε, (the conjunct succeeds);*
  - *if there exists a NAIF-refutation of rank $k' < k$ for $P \cup \{\leftarrow Q'\}$ which does not instantiate $freevar(Q') \setminus \bar{x}$, then $\leftarrow Q$ is a leaf node (the conjunct fails).*

- *The notion of SLDNFI-refutation is obtained by adding in Definition 6.3.2 the condition:*

  (iii) the selected conjunct in G_i is $\forall\bar{x}\neg(Q)$, there exists a finitely failed NAIF-tree of rank $k' < k$ for $P \cup \{\leftarrow Q\}$, $G_{i+1} = G_i \setminus \forall\bar{x}\neg(Q)$ and $\theta_i = \varepsilon$.

In order to prove correctness of NAIF-resolution we show that any NAIF-derivation can be transformed into an "equivalent" SLDNFI-derivation provided that all Γ–*general* queries occurring in the derivation are replaced by some "corresponding" amalgamated queries. This can be done by writing in "detailed" form those amalgamated queries of which the universally quantified negative queries are the abbreviations. We formally define this correspondence as follows:

Definition 6.3.6 *The* only-existential *form of a Γ–general query Q, $\mathcal{E}(Q)$, is inductively defined by*

- *if $Q \equiv A$ then $\mathcal{E}(Q) = A$;*
- *if $Q \equiv \exists\bar{x}\neg Q'$ then $\mathcal{E}(Q) = \exists\bar{x}\neg\mathcal{E}(Q')$;*
- *if $Q \equiv \forall\bar{x}\neg Q'$ then $\mathcal{E}(Q) = \exists\emptyset\neg(\exists\bar{x}\neg(\exists\emptyset\neg\mathcal{E}(Q')))$;*
- *if $Q \equiv Q_1, Q_2$ then $\mathcal{E}(Q) = \mathcal{E}(Q_1), \mathcal{E}(Q_2)$.*

Remark 6.3.7 *It is easy to verify that for all Γ–general query Q, $freevar(Q) = freevar(\mathcal{E}(Q))$.*

Definition 6.3.8 *The* only-existential *form of a Γ–general program P is the amalgamated program, $\mathcal{E}(P)$, obtained by replacing all the bodies, Q, of the clauses of P by their* only-existential *form $\mathcal{E}(Q)$.*

Lemma 6.3.9 *Let P be a Γ–general program and Q a Γ–general query. Then*

- *Q is logically equivalent to $\mathcal{E}(Q)$;*
- *$Comp_L(P)$ is logically equivalent to $Comp_L(\mathcal{E}(P))$.*

Proof Obvious by Definition 6.3.6. □

Upon $\mathcal{E}$-transformation, NAIF-resolution and SLDNI-resolution are equivalent in the following sense:

Proposition 6.3.10 *Let P be a Γ-general program, Q a Γ-general query and $\bar{x} \subseteq freevar(Q)$. Then,*

1. *If $P \cup \{\leftarrow Q\}$ has a NAIF-refutation with c.a.s. ϑ, then $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q)\}$ has an SLDNFI-refutation with c.a.s. ϑ.*
2. *If $P \cup \{\leftarrow Q\}$ has a NAIF-tree which finitely instantiates $\bar{x}$, then $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q)\}$ has an SLDNFI-tree which finitely instantiates $\bar{x}$.*

Proof We reason by mutual generalized induction on the rank k, following the schema of the proof of Theorem 6.2.7.

1. Let $n \geq 0$ be the length of the NAIF-refutation.

   If $n = 0$, then $\leftarrow Q$ is the empty goal.

   Assume now $n > 0$. We distinguish the three cases.

   (i) The selected conjunct is an atom. Let $H \leftarrow Q''$ be the input clause, μ the mgu and $\leftarrow Q'$ the resolvent of $\leftarrow Q$. Then $P \cup \{\leftarrow Q'\}$ has a NAIF-refutation of length $n-1$ with a c.a.s. ϑ' such that $Q\mu\vartheta' = Q\vartheta$. By inductive hypothesis, $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q')\}$ has an SLDNFI-refutation with c.a.s. ϑ'. Then, by selecting in $\mathcal{E}(Q)$ the conjunct $\mathcal{E}(A) = A$, and choosing $H \leftarrow \mathcal{E}(Q'')$ as the input clause, we obtain for $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q)\}$, an SLDNFI-refutation with c.a.s. ϑ. In fact, we have that $mgu(\mathcal{E}(A), H) = mgu(A, H) = \mu$, and, by Remark 6.3.7, $\mathcal{E}(Q') = \mathcal{E}((Q \setminus \{A\} \cup \{Q''\})\mu) = (\mathcal{E}(Q) \setminus \{\mathcal{E}(A)\} \cup \{\mathcal{E}(Q'')\})\mu$.

   (ii) The selected conjunct is a negative query $\exists\bar{y}\neg Q'$, and $P \cup \{\leftarrow Q'\}$ has a NAIF-tree of rank $k' < k$ which finitely instantiates $\bar{y}$. In order to construct an SLDNFI-refutation for $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q)\}$, select in $\mathcal{E}(Q)$ $\mathcal{E}(\exists\bar{y}\neg Q') = \exists\bar{y}\neg\mathcal{E}(Q')$. By inductive hypothesis we have that $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q')\}$ has an SLDNFI-tree of rank $k' < k$ which finitely instantiates $\bar{y}$. Furthermore, $P \cup \{\leftarrow (Q \setminus \exists\bar{y}\neg Q')\}$ has a NAIF-refutation of length $n-1$ with a c.a.s. ϑ. Then, by inductive hypothesis, we have that $\mathcal{E}(P) \cup \{\leftarrow (\mathcal{E}(Q \setminus \exists\bar{y}\neg Q'))\}$ has an SLDNFI-refutation with c.a.s. ϑ. Finally, observe that $\mathcal{E}(Q \setminus \exists\bar{y}\neg Q') = \mathcal{E}(Q) \setminus \mathcal{E}(\exists\bar{y}\neg Q')$.

   (iii) The selected conjunct is a negative query $\forall\bar{y}\neg Q'$, and $P \cup \{\leftarrow Q'\}$ has a finitely failed NAIF-tree of rank $k' < k$. In order to construct an SLDNFI-refutation for $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q)\}$, select in $\mathcal{E}(Q)\}$ $\mathcal{E}(\forall\bar{y}\neg Q') = \exists\emptyset\neg(\exists\bar{y}\neg(\exists\emptyset\neg\mathcal{E}(Q')))$. Now, observe that a finitely failed NAIF-tree is just a NAIF-tree which finitely instantiates $\emptyset$. Thus, by inductive hypothesis we have that $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q')\}$ has an SLDNFI-tree of rank $k' < k$ which finitely instantiates $\emptyset$. This means that the conjunct $\mathcal{E}(\forall\bar{y}\neg Q')$ succeeds under SLDNFI-resolution. Furthermore, $P \cup \{\leftarrow (Q \setminus \forall\bar{y}\neg Q')\}$ has a NAIF-refutation of length $n-1$ with a c.a.s. ϑ. Then, by inductive hypothesis, we have that $\mathcal{E}(P) \cup \{\leftarrow (\mathcal{E}(Q \setminus \exists\bar{y}\neg Q'))\}$ has an SLDNFI-refutation with c.a.s. ϑ. Finally, observe that $\mathcal{E}(Q \setminus \forall\bar{y}\neg Q') = \mathcal{E}(Q) \setminus \mathcal{E}(\forall\bar{y}\neg Q')$.

2. Assume that $P \cup \{\leftarrow Q\}$ has an SLDNFI-tree which finitely instantiates $\bar{x}$ at level h. We have three cases:

   (i) The selected conjunct is an atom A.

   Consider all the children $\leftarrow Q_1, \ldots, \leftarrow Q_n$, $(n \geq 0)$, of $\leftarrow Q$, whose associated substitutions $\vartheta_1, \ldots, \vartheta_n$ satisfy $\neg Inst(\vartheta_i, \bar{x})$. Let $H_i \leftarrow Q'_i$ be the program clause such that $\leftarrow Q_i = \leftarrow (Q \setminus \{A\} \cup \{Q'_i\})\vartheta_i$. By definition, for each $i \in [1,n]$, $P \cup \{\leftarrow Q_i\}$ has a NAIF-tree which finitely instantiates $\bar{x}\vartheta_i$ at level $h_i < h$. By inductive hypothesis, $\forall i \in [1,n]$ $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q_i)\}$ has an SLDNFI-tree which finitely instantiates $\bar{x}\vartheta_i$ at level $h_i < h$. Finally, observe that by Definition 6.3.6 and Remark 6.3.7, for all $i \in [1,n]$, $\mathcal{E}(Q_i) = \mathcal{E}((Q \setminus \{A\} \cup \{Q'_i\})\vartheta_i) = (\mathcal{E}(Q) \setminus \{\mathcal{E}(A)\} \cup \{\mathcal{E}(Q'_i)\})\vartheta_i$.

   (ii) The selected conjunct is a negative query $\exists\bar{y}\neg Q'$. Consider the two cases:

   (a) $P \cup \{\leftarrow Q'\}$ has a NAIF-refutation of rank $k' < k$ which does not instantiate *freevar*(Q'). By inductive hypothesis and Remark 6.3.7, we have that $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q')\}$ has an SLDNFI-refutation of rank $k' < k$ which does not instantiate *freevar*$(\mathcal{E}(Q'))$.

   (b) $P \cup \{\leftarrow Q'\}$ has a NAIF-tree of rank $k' < k$ which finitely instantiates $\bar{y}$, so the (only) child of $\leftarrow Q$ is $\leftarrow (Q \setminus \exists\bar{y}\neg Q')$. By hypothesis, $P \cup \{\leftarrow (Q \setminus \exists\bar{y}\neg Q')\}$ has a NAIF-tree of rank $k'' < k$ which finitely instantiates $\bar{x}$. By inductive hypothesis we have that

   - $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q')\}$ has an SLDNFI-tree of rank $k' < k$ which finitely instantiates $\bar{y}$; since by Definition 6.3.6, $\mathcal{E}(\exists\bar{y}\neg Q') = \exists\bar{y}\neg\mathcal{E}(Q')$, the (only) child of $\leftarrow \mathcal{E}(Q)$ is $\leftarrow (\mathcal{E}(Q) \setminus \mathcal{E}(\exists\bar{y}\neg Q'))$; and
   - $\mathcal{E}(P) \cup \{\leftarrow (\mathcal{E}(Q \setminus \exists\bar{y}\neg Q'))\}$, i.e. $\mathcal{E}(P) \cup \{\leftarrow (\mathcal{E}(Q) \setminus \mathcal{E}(\exists\bar{y}\neg Q'))\}$ has an SLDNFI-tree of rank $k'' < k$ which finitely instantiates $\bar{x}$.

   (iii) The selected conjunct is a negative query $\forall\bar{y}\neg Q'$. Consider the two cases:

   (a) $P \cup \{\leftarrow Q'\}$ has a NAIF-refutation of rank $k' < k$ which does not instantiate *freevar*$(Q') \setminus \bar{y}$. By inductive hypothesis and Remark 6.3.7, we have that $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q')\}$ has an SLDNFI-refutation of rank $k' < k$ which does not instantiate *freevar*$(\mathcal{E}(Q')) \setminus \bar{y}$.

   (b) $P \cup \{\leftarrow Q'\}$ has a finitely failed NAIF-tree of rank $k' < k$, so the (only) child of $\leftarrow Q$ is $\leftarrow (Q \setminus \forall\bar{y}\neg Q')$. By hypothesis, $P \cup \{\leftarrow (Q \setminus \forall\bar{y}\neg Q')\}$ has a NAIF-tree of rank $k'' < k$ which finitely instantiates $\bar{x}$. By inductive hypothesis we have that

   - $\mathcal{E}(P) \cup \{\leftarrow \mathcal{E}(Q')\}$ has a finitely failed SLDNFI-tree of rank $k' < k$ (i.e. an SLDNFI-tree of rank $k' < k$ which finitely instantiates $\emptyset$), so the conjunct
     $$\mathcal{E}(\forall\bar{y}\neg Q') = \exists\emptyset\neg(\exists\bar{y}\neg(\exists\emptyset\neg\mathcal{E}(Q')))$$
     succeeds under SLDNFI-resolution and the (only) child of $\leftarrow \mathcal{E}(Q)$ is $\leftarrow (\mathcal{E}(Q) \setminus \mathcal{E}(\forall\bar{y}\neg Q'))$; and
   - $\mathcal{E}(P) \cup \{\leftarrow (\mathcal{E}(Q \setminus \forall\bar{y}\neg Q'))\}$, i.e. $\mathcal{E}(P) \cup \{\leftarrow (\mathcal{E}(Q) \setminus \mathcal{E}(\forall\bar{y}\neg Q'))\}$ has an SLDNFI-tree of rank $k'' < k$ which finitely instantiates $\bar{x}$.

$\square$

Theorem 6.3.11 (Correctness of NAIF-resolution) *Let P be a Γ−general program, $\leftarrow Q$ a Γ−general goal and $\bar{x} \subseteq$ freevar(Q). Then*

1. *if* $P \cup \{\leftarrow Q\}$ *has a NAIF-refutation with c.a.s.* ϑ, *then* $Comp_L(P) \models Q\vartheta$,

2. *if* $P \cup \{\leftarrow Q\}$ *has a NAIF-tree which finitely instantiates* $\bar{x}$, *then* $Comp_L(P) \models \exists\bar{x}\neg Q$.

3. *if* $P \cup \{\leftarrow Q\}$ *has a finitely failed NAIF-tree, then* $Comp_L(P) \models \forall\bar{x}\neg Q$.

Proof Points 1. and 2. follow by Proposition 6.3.10, Theorem 6.3.3 and Lemma 6.3.9. For the point 3. observe that it is obtained from point 2. with $\bar{x} = \emptyset$. □

We end this section with some examples showing how several problems can be naturally expressed in our language in a form very similar to their specification.

Example 6.3.12 *In logic programming sets are usually represented as lists, and the "membership" relation is defined as*

$$member(x, [x|y]) \leftarrow,$$
$$member(x, [z|y]) \leftarrow member(x, y).$$

Consider now the subset relation, formally expressed by

$$Y \subseteq Z \quad \Leftrightarrow \quad \forall x.(x \in Y \Rightarrow x \in Z)$$
i.e.
$$Y \subseteq Z \quad \Leftrightarrow \quad \forall x.\neg(x \in Y \wedge x \notin Z)$$

In our language this relation can be correctly defined as

$$subset(y, z) \leftarrow \forall x.\neg(member(x, y), \neg member(x, z)).$$

This definition is "complete" w.r.t. the positive use, in the sense that, given two ground lists t, u *which represent two sets* T, U *with* $T \subseteq U$, *then the query* $\leftarrow subset(t, u)$ *has a NAIF-refutation. It is also complete w.r.t. the negative use, in the sense that if* $T \not\subseteq U$, *then the query* $\leftarrow \neg subset(t, u)$ *succeeds, because* $\leftarrow subset(t, u)$ *has a finitely failed NAIF-tree.*

Note that if we eliminate "$\forall x$*" from the definition, then it is not complete in the negative use anymore.*

Example 6.3.13 *The set-disjunction relation, formally expressed by*

$$Y \cap Z \quad \Leftrightarrow \quad \forall x.(x \in Y \Rightarrow x \notin Z)$$
i.e.
$$Y \cap Z \quad \Leftrightarrow \quad \forall x.\neg(x \in Y \wedge x \in Z),$$

in our language can be correctly defined as

$$disjunct(y, z) \leftarrow \forall x.\neg(member(x, y), member(x, z)).$$

One can easily verify that also this definition is complete in both the negative and the positive use.

Example 6.3.14 *Consider a binary operation op, defined in Logic Programming by a ternary relation* p_{op} *(i.e.* $p_{op}(x, y, z) \Leftrightarrow op(x, y) = z$*). The neutral element, formally defined as*

$$neutral(x) \Leftrightarrow op(x, y) = y \wedge op(y, x) = y$$

can be defined in our language by

$$neutral(x) \leftarrow \neg\exists y.\neg(p_{op}(x,y,y), p_{op}(y,x,y)).$$

The opposite can be defined as

$$not\_neutral(x) \leftarrow \neg neutral(x)$$

or, more simply

$$not\_neutral(x) \leftarrow \exists y.\neg p_{op}(x,y,y),$$
$$not\_neutral(x) \leftarrow \exists y.\neg p_{op}(y,x,y).$$

Observe the nested use of quantifiers in both Examples 6.3.12 and 6.3.14 of the form, respectively, $\forall\bar{x}\neg(\ldots\exists\bar{y}\neg(\ldots))$ with $\bar{y}=\emptyset$, and $\exists\bar{x}\neg\ (\ldots\exists\bar{y}\neg(\ldots))$ with $\bar{x}=\emptyset$. The next shows how more complicated relations as the Smyth preordering can be expressed by nesting the universal quantifiers.

Example 6.3.15 *The Smyth preordering relation is formally expressed by*

$$Y \preceq_S Z \Leftrightarrow \forall x \in Z.\ \exists w \in Y.\ w \leq x$$

where "$\leq$" is a partial order on sets which suppose already defined. In our language we can define it by

$$Smyth(y,z) \leftarrow\ \forall x.\neg(member(x,z), \forall w.\neg(member(w,y), leq(w,x))).$$

Example 6.3.16 *This example shows the limits of NAIF-resolution deriving from its inability to compute bindings.*

The implication predicate can be formally defined as

$$x \supset y \Leftrightarrow\ y \text{ is true whenever } x \text{ is true.}$$

A $\Gamma-$general program defining it could be:

$$\begin{aligned} P \quad = \quad \{ \ & Implies(x,y) \leftarrow \neg(true(x), false(y)), \\ & true(T) \leftarrow, \\ & false(F) \leftarrow\}. \end{aligned}$$

If we use the program P to check whether $t \supset u$ holds for t and u being one of the truth values T or F, then it is complete both in the positive and in the negative use. For instance, $\leftarrow Implies(T,T)$ succeeds, and $\leftarrow Implies(T,F)$ fails. However, if our aim is to find values x and y for which $Implies(x,y)$ is true (or false), then we don't obtain any answers because of the dead end.

Correctness for extended programs

From previous examples we can argue that $\Gamma-$*general* programs are more expressive than general programs, in the sense that they allow to derive programs from their logical specification quite easily. Extended programs, whose clauses can have as body any first-order formula, are even more expressive. For these programs Lloyd and Topor have shown that SLDNF-resolution is correct, by translating them into general programs ([72]). In a similar way, we show that NAIF-resolution can be correctly applied to extended programs previously transformed into

$\Gamma-$*general* programs. This transformation is very simple, so the correctness result derives immediately from Theorem 6.3.11.

An *extended program* is a finite set of first-order formulas of the form $A \leftarrow W$, where A is an atom and W is a first-order formula.

An *extended goal* is a first-order formula of the form $\leftarrow W$, where W is a (not necessarily closed) first-order formula.

Note that SLDNF and SLDNI-resolution only allow program clauses of the form $\leftarrow W$, where W is respectively a conjunction of literals and a conjunction of $\exists\neg$*general* queries. We will see that every first-order formula is logically equivalent to a $\Gamma-$*general* query.

Next we give the definition of completion of an extended program P w.r.t. the language.

Definition 6.3.17 *Let P be an extended program. Its completion w.r.t. the language L is defined in the usual way, namely $Comp_L(P) = IFF(P) \cup CET_L$, where $IFF(P)$ is the collection of completed definitions of predicates in P, i.e. the set of formulas*

$$p(\bar{x}) \leftrightarrow \bigvee_{i \in [1,s]} E_i,$$

where $\bar{x} = x_1, \ldots, x_n$ are variables not appearing anywhere in the definition of p, and, for all i s, if the i-th clause in P with p in the head is of the form $p(\bar{t_i}) \leftarrow W_i$, then E_i is the formula $\exists \bar{y_i}.(\bar{x} = \bar{t_i} \wedge W_i)$, where $\bar{y_i}$ denotes the variables in $p(\bar{t_i})$ and the free variables in W_i.

Definition 6.3.18 *Given two formulas W_1 and W_2 of a first order language $\mathcal{L}$ we say that W_1 is logically equivalent to W_2 if $\forall(W_1 \leftrightarrow W_2)$ is valid, i.e. it is true in every interpretation of $\mathcal{L}$.*

Proposition 6.3.19 *For each first-order formula W, there is a $\Gamma-$general query Q such that W and Q are logically equivalent.*

Proof (By induction on the structure of W)

($W = A$) Obvious.

($W = \neg W'$) By inductive hypothesis there exists a $\Gamma-$*general* query Q' such that W' and Q' are logically equivalent, then also $\neg W'$ and $\neg Q'$ are logically equivalent. Finally observe that $\neg Q'$ is equivalent to the $\Gamma-$*general* query $\exists\emptyset.\neg Q'$.

($W = W_1 \wedge W_2$) By inductive hypothesis there exist $\Gamma-$*general* queries $Q = Q_1, Q_2$ logically equivalent to W_1 and W_2. Then $Q \equiv Q_1, Q_2$ is a $\Gamma-$*general* query and it is still logically equivalent to W.

($W = W_1 \vee W_2$) Observe that W is logically equivalent to $W' = \neg(\neg W_1 \wedge \neg W_2)$. Then apply the previous cases and the transitivity of the logical equivalence relation.

($W = W_1 \rightarrow W_2$) Observe that W is logically equivalent to $W' = \neg(W_1 \wedge \neg W_2)$. Then apply the previous cases.

($W = W_1 \leftrightarrow W_2$) Observe that W is logically equivalent to $W' = (W_1 \rightarrow W_2) \wedge (W_2 \rightarrow W_1)$. Then apply the previous cases and the transitivity of the logical equivalence relation.

($W = \exists W'$) By inductive hypothesis there exists a $\Gamma-$*general* query Q' logically equivalent to W'. Now observe that the following is a sequence of logically equivalent formulas:

$$W, \neg(\neg(\exists W')), \neg(\forall\neg W'), \neg(\forall\neg Q'),$$

and that $\neg(\forall\neg Q')$ is a $\Gamma-$*general* query.

($W = \forall W'$) Observe that W is logically equivalent to $W'' = \neg(\exists\neg W')$. Then apply the previous cases and the transitivity of the logical equivalence relation.

□

Definition 6.3.20 *Let P be an extended program and G an extended goal. The $\Gamma-$general forms of P and G are the $\Gamma-$general program P' and the $\Gamma-$general goal G' obtained by substituting in P and in G all the first-order formulas by the corresponding logically equivalent $\Gamma-$general query.*

Lemma 6.3.21 *Let P be an extended program and P' its $\Gamma-$general form. Then P and P' are logically equivalent and also $Comp_L(P)$ and $Comp_L(P')$ are logically equivalent.*

Proof Straightforward. □

Lemma 6.3.22 *Let P be an extended program, P' its $\Gamma-$general form and F a formula containing only predicates which appear in P. Then*

If $Comp_L(P') \models F$ then $Comp_L(P) \models F$.

Proof Obvious by Lemma 6.3.21 □

We now define the interpreter for extended programs. It is just NAIF-resolution as in Definition 6.3.5, applied to the $\Gamma-$*general* forms of the extended programs.

Definition 6.3.23 *Let P be an extended program, G an extended goal and R a selection rule. The NAIF-resolution for $P \cup \{G\}$ is the NAIF-resolution for $P' \cup \{G'\}$, where P' and G' are the $\Gamma-$general forms of P and G.*

Theorem 6.3.24 (Correctness of NAIF-resolution for extended programs) *Let P be an extended program, $\leftarrow W$ an extended goal and $\bar{x} \subseteq freevar(W)$. Then*

1. *if $P \cup \{\leftarrow W\}$ has a NAIF-refutation with c.a.s. ϑ, then $Comp_L(P) \models W\vartheta$,*

2. *if $P \cup \{\leftarrow W\}$ has a NAIF-tree which finitely instantiates $\bar{x}$, then $Comp_L(P) \models \exists\bar{x}\neg W$.*

3. *if $P \cup \{\leftarrow W\}$ has a finitely failed NAIF-tree, then $Comp_L(P) \models \forall\bar{x}\neg W$.*

Proof By Theorem 6.3.11 and Lemma 6.3.22. □

Comparison with related works

The idea of using extended programs to solve general programming problems is not new. Clark himself, when introduced Negation As Failure, discussed about the transformation of extended programs into general programs (see [23] p.299), and a version of it has been implemented in micro-Prolog by Clark and McCabe, ([21]).

On the other hand, in [64] Kowalski asserted that " any problems can be converted from standard form to clausal form", (where standard form means the formalism using the full first order logic), and that "useful inference system for the standard form of logic may be obtained by combining inference rules for clausal form with rules for converting from standard to clausal form".

This is what Lloyd and Topor have done in [72], where a system of rules transforming an extended program P into a general program P' is given, and SLDNF-resolution is applied to the transformed programs. In this way, they prove correctness of SLDNF-resolution for extended programs and goals, justifying the introduction of them in logic programming by the fact that, for a given problem, an extended program is usually clearer and simpler than the corresponding general program because it is in a form similar to the (logical) specification of the problem.

The same reasons motivate our arrangement of NAIF-resolution to compute with extended programs. Anyway, our translation is more simple and immediate than the transformation used by Lloyd and Topor. We merely substitute verbatim in every extended clause $A \leftarrow W_1, \ldots, W_n$ all the formulas W_i in a logically equivalent $\Gamma-$*general* query, thus obtaining a $\Gamma-$*general* program. No clauses are added to the programs. On the transformed program we use NAIF-resolution instead of SLDNF-resolution. This makes our system more powerful than the system based on SLDNF-resolution (floundering does not occur).

Example 6.3.25 *Consider the extended program*

$$P = \{\ p(a) \leftarrow .\ \ \} \tag{6.1}$$

with the extended goal $G = \leftarrow \forall x.p(x)$.

The $\Gamma-$*general form of* G *is* $G' = \leftarrow \neg\exists x.\neg p(x) \equiv \leftarrow Q$. *We have that* $P \cup \{G'\}$ *fails in the NAIF system, and in fact* $Comp_L(P) \models \neg Q$, *i.e.* $Comp_L(P) \models \exists x.\neg p(x)$. *On the other hand, the translation of* G *in a general form, as done by Lloyd and Topor is* $G'' = \leftarrow answer()$, *with*

$$answer() \leftarrow \neg q(),$$

and

$$q() \leftarrow \neg p(x)$$

added to P *to obtain* P'', *and SLDNF-resolution flounders with* $P'' \cup \{G''\}$.

Example 6.3.26 *Imagine a scene in which some blocks A, B, C and D are disposed in a way such that D is over B and C, which are over A. Consider the problem to show that the block A stands under all the others. The situation can be described by the following extended program and goal:*

$$\begin{array}{rcl} P & = & \{\ Over(x, A) \leftarrow, \\ & & \quad Over(D, B) \leftarrow, \\ & & \quad Over(D, C) \leftarrow, \\ & & \quad Covered(y) \leftarrow \forall x.Over(x, y) \quad \} \\ G & = & \leftarrow Covered(A). \end{array}$$

The Γ–general form of P is

$$\begin{array}{rl} P' = & (P \setminus \{Covered(y) \leftarrow \forall x.Over(x,y)\}) \cup \\ & \{Covered(y) \leftarrow \neg\exists x.\neg Over(x,y)\}. \end{array}$$

We have that $P' \cup \{G\}$ succeeds in the NAIF system, and in fact $Comp_L(P) \models Covered(A)$. On the other hand, the translation of P in a general program P'', as done by Lloyd and Topor, is obtained by substituting in P the extended clause $Covered(y) \leftarrow \forall x.On(x,y)$ by the two general clauses $Covered(y) \leftarrow \neg p(y)$ and $p(y) \leftarrow \neg On(x,y)$, and SLDNF-resolution flounders with $P'' \cup \{G\}$.

Several examples of extended program clauses can be found in Kowalski's book (see Chapter 10 in [64]). We end this section by translating some of them to show the simplicity and immediacy of doing it.

Example 6.3.27 *([64] p.203) We want express that " Bob is happy if all his students like logic". The standard form is*

$$Happy(Bob) \leftarrow \forall x.(Studentof(Bob,x) \rightarrow Likes(x,logic)).$$

The Γ–general form is

$$Happy(Bob) \leftarrow \forall x.\neg(Studentof(Bob,x), \neg Likes(x,logic)).$$

Example 6.3.28 *([64], p.203) A set is well-founded iff every non-empty subset has a least element. A set is non-empty iff it has at least one element. An element of a set is a least element iff it is less than or equal to every element of the set.*
The standard form is

$$\begin{array}{rcl} Wellorderedset(x) & \leftrightarrow & \forall z.(Hasleastelmt(z) \leftarrow subset(z,x) \wedge Nonempty(z)) \\ Nonempty(z) & \leftrightarrow & \exists u.member(u,z) \\ Hasleastelemt(z) & \leftrightarrow & \exists u.[member(u,z) \wedge \forall v.(member(v,z) \rightarrow leq(u,v))]. \end{array}$$

The Γ–general form is

$$\begin{array}{rcl} Wellorderedset(x) & \leftarrow & \forall z.\neg(subset(z,x), Nonempty(z), \neg Hasleastelmt(z)) \\ Nonempty(z) & \leftarrow & member(u,z) \\ Hasleastelemt(z) & \leftarrow & member(u,z), \forall v.\neg((member(v,z), \neg leq(u,v))). \end{array}$$

Note that the completed definitions of P are exactly the standard form given above.

Example 6.3.29 *Finally, we give the steps needed in the system of Lloyd and Topor to transform the extended clause defining the subset relation*

$$C \equiv \;\; subset(y,z) \leftarrow \forall x.(member(x,y) \rightarrow member(x,z)).$$

into a general clause.

step 1. $subset(y,z) \leftarrow \forall x.\neg(member(x,y) \wedge \neg member(x,z)).$

step 2. $subset(y,z) \leftarrow \neg\exists x.(member(x,y) \wedge \neg member(x,z)).$

step 3. $subset(y,z) \leftarrow \neg p(y,z)$
and
$p(y,z) \leftarrow \exists x.(member(x,y) \wedge \neg member(x,z))$.

step 4. $subset(y,z) \leftarrow \neg p(y,z)$
and
$p(y,z) \leftarrow (member(x,y), \neg member(x,z))$.

Observe that in our system only step 1 is required to obtain the Γ–general form of C (see Example 6.3.12), provided that "$\wedge$" is changed in ",".

6.3.3 Incompleteness

The two amalgamations we have presented are not complete, as one might expect, since they result from the combination of two incomplete methods. In particular, both SLDNFI-resolution and NAIF-resolution suffer from the dead end problem as well as SLDNI-resolution. This problem is however limited to the case in which there is a refutation instantiating the free variables of a selected negative conjunct. Obviously, to the purpose of approximating completeness as much as possible, it is better to delay the selection of a negative conjunct until all the positive literals have been resolved, and all possible bindings generated.

Example 6.3.30 *Consider the program*

$$\begin{aligned} P = \ & \{p(x) \leftarrow \exists\emptyset\neg q(x,y), r(y), \\ & q(x,a) \leftarrow, \\ & r(a) \leftarrow\}. \end{aligned}$$

We have $Comp_L(P) \models \neg p(x)$, but, due to the local variable y, if we select $\exists\emptyset\neg q(x,y)$ before $r(y)$, then neither SLDNFI-resolution nor NAIF-resolution are able to derive the success of $P \cup \{\leftarrow q(x,y)\}$ (they come to a dead end).

Thanks to the ability to deal with existentially quantified conjuncts, hence with local variables, the amalgamated treatment of negation fits better than negation as failure to be the operational counterpart of the completion semantics.

Example 6.3.31 *Consider the program*

$$\begin{aligned} P = \ & \{p(x) \leftarrow \neg q(x,y), \\ & q(x,a) \leftarrow, \\ & r(b) \leftarrow\}. \end{aligned}$$

We have $Comp(P) \models p(x)$, but, due to the local variable y, the standard SLDNF-resolution would not derive the success of $P \cup \{\leftarrow p(x)\}$, not even when modified according to Definition 6.3.2 or Definition 6.3.5.

On the other hand, in the amalgamated language, as well as in the Γ-general language P would be naturally written as

$$\begin{aligned} P' = \ & \{p(x) \leftarrow \exists y \neg q(x,y), \\ & q(x,a) \leftarrow, \\ & r(b) \leftarrow\} \end{aligned}$$

and $P' \cup \{\leftarrow q(x)\}$ has an SLDNFI-refutation (which is also a NAIF-refutation) with empty substitution.

6.4 Beyond the completion: a semantics survey

Clark's completion is undoubtedly the more widely used formalism. Its main advantage is that of computability (the set of its logical consequences is recursively enumerable), which allows to obtain effectivity of the interpreter and to establish results such as the completeness of the NAF. This property is not enjoyed, in general by the declaratively-driven semantics where the interpreter has only a theoretical significance and a stronger rule than NAF (read "negation as finite failure") would be required for completeness. On the other hand, the effectivity is achieved in the completion semantics for the extreme specialization of the notion of completion which has no relevance in areas other than logic programming. In other words, $Comp(P)$ is strongly tied to the procedural reading of a negated formula under the scheme "observable (i.e. finite) failure of SLD-resolution to prove", so that it is not sufficiently general to be applied outside logic programming. In particular, it does not play a major role in formalizing non-monotonic reasoning in AI. On the other hand, studying the relations between non-monotonic reasoning and logic programming brings to benefits to both areas by yielding, in a direction, a deeper insight into the semantics of negation as failure, and, in the opposite direction, reasoning methods based on logic programming techniques for the major non-monotonic formalisms and new forms of non-monotonic reasoning.

A larger number of recent works have been devoted to this relationship and have brought to the definition of new semantics for logic programs, leading to interpretations of negation as failure not consonant in general with the NAF of the completion semantics.

However, no single form of negation seems to be completely satisfactory for all applications. As a matter of fact, it is always possible to find an example where a given semantics do not have the desired behavior. This fact justifies the never ending proliferation of semantics.

6.4.1 Minimal model semantics

There is a clear correlation between the notion of *minimality* in a partial order and negation as failure:[2] if $M \subset N$, then every atom which is false in N is also false in M. Thus M seems to come closer to exhibiting negation as failure than N does. Therefore, the set of all minimal Herbrand models represent a reasonable (and often preferable) alternative to the least Herbrand model semantics, particularly useful when we have to do with general programs, since for these a least Herbrand model usually does not exist.

By *minimal model semantics* it is meant the semantics induced by the set MIN(P) of all minimal models of the program P. Under this semantics a formulas F is considered to be true if and only if it is satisfied in all minimal models of P.

The idea of replacing the notion of least model by that of minimal model was suggested by Minker ([76]), in the context of deductive databases, in order to solve the problem of the inconsistency of cwa semantics for programs implying indefinite information. In fact, cwa is consistent only if the program has a least Herbrand model. The typical example is the program

$$P: \; p \leftarrow \neg q,$$

whose models are $\{p\}, \{q\}$ and $\{p, q\}$. Since P has not a least Herbrand model, cwa is inconsistent. However, $\{p\}$ and $\{q\}$ are both minimal models and represent the *logical meaning* of P, i.e. $p \vee q$. However, negation cannot be now interpreted by SLDNF-resolution, since the latter would treat q as false. The problem is that minimal model semantics does not introduce any

[2] Recall that Herbrand models are identified with subsets of the Herbrand base. Thus Herbrand models are naturally partially ordered by $\subseteq$.

new information, whereas, due to the presence of negative premises in a general program, the semantics should imply also the positive information deriving from the negation of the positive atoms which are not provable from the program.

The reformulation of cwa in terms of minimal models is known as *generalized closed world assumption* (gcwa), and it is always consistent. For the reasons above mentioned, SLDNF-resolution is not correct w.r.t. gcwa, so other proof procedures has been contrived which are more appropriate to it (see for instance [52]).

In order to find a more satisfactory semantics for general programs, Apt Blair and Walker ([5]) suggested that the models of a given program P to be studied were those (Herbrand) models which are not only minimal, but even *supported*, i.e. a ground atom A is true only if there is a ground instance of a clause of P with head A and body which is true. This idea has revealed itself particularly useful in connection with the class of stratified programs, for which the existence of a unique "natural" model which is both minimal and supported (the so-called perfect model) can be proved.

6.4.2 Perfect models of stratified programs

On the class of *stratified programs* there is a well-accepted semantics, called the perfect model semantics. Stratified programs and this semantics were introduced in [5] and [44], with earlier related works in [19]. Later researchers ([84]) have applied the same semantics to larger classes of programs, and shown that it is equivalent to suitable forms of the major formalizations of non-monotonic reasoning in AI. In any case, many interesting logic programs are stratified; in fact, for a time it was suspected by many that all interesting logic programs are stratified. Like the van Emden-Kowalski semantics, the perfect model semantics for stratified programs picks out a *unique* Herbrand model, called the perfect model, as the intended model of the program. This property is not enjoyed by the other semantics discussed in this section nor by completion semantics. Like the least Herbrand model in the van Emden-Kowalski semantics, the perfect model is widely accepted as capturing exactly the intuitions of negation as failure.

A stratified program is a program in which a predicate symbol can be recursively defined provided that negation is not involved into recursion. Thus, stratification is a generalization of the concept of the *hierarchical constraint* of [23] which rules out any form of recursive definition. Equivalently, a general program P is stratified if it is the union

$$\mathcal{P}_1 \cup \mathcal{P}_2 \cup \cdots \cup P_n$$

of programs where, (1) each predicate symbol is defined in at most one of the $\mathcal{P}_i$'s, and (2) for $1 \leq i \leq n$, each predicate symbol defined in $\mathcal{P}_i$ is used positively only in $\mathcal{P}_j$'s for $j \geq i$ and used negatively only in $\mathcal{P}_j$'s for $j > i$. The sequence $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n)$ is a *stratification* of $\mathcal{P}$, and $\mathcal{P}_1, \mathcal{P}_2, \ldots \mathcal{P}_n$ are *strata*.

For a general program more than a single stratification is possible. Note that, in particular, if $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n)$ is a stratification of $\mathcal{P}$, then $\mathcal{P}_1$ is a definite program.

For a stratified program it is possible to construct a model in a way which is vary similar to the construction of the least Herbrand model of a positive program. Here we use the immediate consequence operator T_P associated to the program P; there a similar operator T_{P_i} is considered on each stratum P_i of a stratification of the program. Of this operator *cumulative* powers have to be considered to take into account non-monotonicity and the model M_P is constructed by taking for all $2 \leq i \leq n$ the least fixpoint of T_{P_i} starting from the set corresponding to the least fixpoint of $T_{P_{i-1}}$ and the least fixpoint of T_{P_1} starting from the empty set. The claim that M_P is the intended (or natural) model of P is supported by the fact that, even though its construction apparently depends on the strata of P, it is actually independent from the particular stratification considered. This is because stratified programs admit a unique *perfect* Herbrand model which is equivalent to M_P, no matter which stratification is considered. The notion of perfect model is referred to a *preference* relation between models induced by a well-founded ordering on the Herbrand base establishing a *priority* between atoms. A model is called perfect if there are no models preferable to it.

As already mentioned at the beginning, perfect models semantics extends significantly beyond the class of stratified programs, and also non-Herbrand perfect models are considered in general. However, only for stratified logic programs it enjoys the property of "uniqueness". Moreover, for stratified logic programs (suitable forms of) the four major formalizations of non-monotonic reasoning in AI (McCarthy's circumscription, Reiter's closed world assumption, Moore's autoepistemic logic and Reiter's default theory) all coincide with the perfect model semantics ([84]).

6.4.3 The stable and well-founded semantics

The stable and well-founded semantics are rather similar to the program completion semantics but capture an intuition that the program completion semantics does not: that a ground atom may be inferred if and only if it is provable in a finite number of steps from negated atoms which are true (or derived).

Stable semantics has been inspired by autoepistemic logic, and in [45] the notion of stable model is just a paraphrasing in terms of logic programming concepts of the "stable expansions" in that logic. We give the intuition behind the definition of the stable models of a program P. If M is a set of ground atoms that are considered to be true, then any clause in P that has a subgoal $\neg B$ with $B \in M$ is useless; furthermore, any subgoal $\neg B$ with $B \notin M$ is trivial. Therefore, an Herbrand interpretation M is defined a stable model if and only if it happens to be precisely the set of atoms that are logical consequences of the transformed program P_M obtained by

- eliminating every clause whose body contains a negative literal $\neg B$ with $B \notin M$;
- deleting all negative literals from the body of the remaining clauses.

There are programs which have no stable models, like $\{p \leftarrow \neg p\}$. Moreover, stable models may not be unique. For example, $\{p \leftarrow \neg q,\ q \leftarrow \neg p\}$ has two stable models: $\{p\}$ and $\{q\}$.

Differently from stable model semantics, well-founded semantics was, originally, not clearly related to commonsense reasoning, but was later compared with several non-monotonic formalisms. It has been introduced in [106] as an extension of perfect models semantics by taking 3-valued logic into account. From a comparison between stable models semantics and well-founded semantics it turns out that the former is more general: if a program has a well-founded model, then that model is its unique stable model. Stratified program have a well-founded semantics and then a unique stable model, and they are equivalent to the perfect model semantics.

Both the stable and well-founded semantics have proved to be robust concepts, having a variety of equivalent constructions. There are second order logic characterizations by Van Gelder ([107]). There are extensive developments of these and related semantics in terms of negations as hypotheses, argumentation, and relationships with abductive inference; this has been developed in several papers, e.g. [37, 36, 62, 13].

Chapter 7

Conclusions

7.1 Summing up

We summarize the main results of this thesis by collecting them in the following points.

- We have defined a failure set (the *FFI* set) which is the negative counterpart of the atomic consequences set of a program. This set is operationally characterized as the set of atoms for which every SLD-derivation either fails or instantiates some variables. Then, we have shown that *FFI* is the set of atoms whose existentially quantified negation is a logical consequence of the completion of the program P. This model-theoretic characterization has given the foundations of a new rule for the treatment of negated atoms, the Negation As Instantiation rule, which infers $\exists\neg A$ if $P \cup \{\leftarrow A\}$ has a "finitely instantiating" SLD-tree. The NAI rule is as easy to implement as the NAF rule and it is orthogonal to it. Then, we have defined an interpreter for logic programs and goals which is able to handle existentially quantified negative subgoals by using both NAI and NAF. We have called such an interpreter SLDNFI-resolution and we have proved its correctness. Another interpreter, the NAIF-resolution results from a slight modification of SLDNFI-resolution. It is able to deal with both existentially and universally quantified negative queries (the $\Gamma-$*general* queries). We have shown that $\Gamma-$*general* programs are as expressive as extended programs by exhibiting a translation of the latter into the former. Then, we have proved correctness of NAIF-resolution for extended programs by first transforming them into $\Gamma-$*general* programs and then using the fact that NAIF-resolution is correct in this case. Our translation is more immediate than the mechanism introduced by Lloyd and Topor to translate extended programs into general ones.

- We have proposed a solution to the problem of characterizing the infinite computations of the programs by defining a model for a general constraint logic programming language which we have called Γ_clp language. This model is obtained by using techniques which are typically adopted for defining the semantics of more conventional programming languages. In particular, we have defined a structural operational semantics based on a transition system, by means of which we have introduced a notion of observables for a goal G, $(\mathcal{O}_P(G))$, which collects the results of the successful, failed and infinite computations of G from a given program P. Then, we have defined a denotational semantics, $\mathcal{G}_P$, in the Scott-Strackey style, and we have shown that it is correct and fully abstract with respect to the observables $\mathcal{O}_P$. If restricted to the terminating computations, this model corresponds to the C-semantic for logic programming. We have also shown that the denotation of a goal G, $\mathcal{G}_P[\![G]\!]$, is indeed an element of the Smyth powerdomain of the constraint system underlying the language. Another kind of powerdomain construction should get nearer the S-semantics, namely the Plotkin powerdomain.

- We have considered the possibility to introduce negation in the language Γ_clp. Due to the non-monotonicity of this construct, it has been not possible to extend the model $\mathcal{G}_P$ to a language allowing the use of negated goals into recursion. Thus, we have restricted ourselves to the case of negative goals, $\neg G$, which are computed from positive programs. This restriction has allowed us to define a simple model for $\neg G$. This is obtained from the corresponding model for G by using the notion of Heyting negation. Finally, we have conjectured some possibilities to define a denotational semantics for the language extended with a restricted use of negation (double negation or stratification).

7.2 Future work

Our plans for future work include the investigation of the problem of infinite computations for the concurrent constraint programming (ccp) paradigm as developed in [89]. As already mentioned, the language Γ_clp we have defined in Section 4.2.2, is a special case (no synchronization is present) of a sublanguage of ccp, namely the Angelic ccp ([59]). Here the task is made easier by the presence of a *don't know* non-determinism. Our objective is to develop an as much simple model for the input/output observables in ccp. In doing so, the combination of non-determinism and infinite computations represents a serious obstacle due to the presence of a *don't care* non-determinism. As discussed in Section 4, a powerdomain approach does not work in this case. Also a metric approach seems to be not appropriate. We think that the most promising perspectives are offered by the category theory.

Another objective of our future research is to find out if a suitable interpretation of the synchronization construct, (*ask* process) of the Angelic ccp could be found in the cHa's theory. In this thesis we didn't exploit completely the possibilities offered by these structure. In fact, we have used only the negation operation, whereas cHa's admit also implication. We think that the last operation could allow an intuitionistic interpretation of the ask operator in Angelic ccp, thus improving the semantic for this construct proposed in [59, 68], which is essentially based on the classical logic.

Bibliography

[1] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[2] R. Aczel. *Non-well-founded sets*, volume 14 of *Lecture Notes*. Stanford University, 1988.

[3] H. Andreka, M. H. van Emden, I. Nemeti, and J. Tiuryn. Infinite-term semantics for logic programs. Draft manuscript, 1983.

[4] K. R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[5] K. R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, Ca., 1988.

[6] K. R. Apt and G. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, October 1986.

[7] K. R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[8] A. Arnold and M. Nivat. The Metric Space of Infinite Trees: Algebraic and Topological Properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

[9] C. Baral, J. Lobo, and J. Minker. Generalised disjunctive well-founded semantics for logic programs. In *Proc. ISMIS'90*, pages 465–473, October 1990.

[10] R. Barbuti, P.Mancarella, D. Pedreschi, and F. Turini. A trasformational approach to negation in logic programming. *Journal of Logic Programming*, 8:201–228, 1990.

[11] A. Batarekh and V. Subrahmanian. The query topology in logic programming. In B. Monien and R. Cori, editors, *Proc. STACS'89*, volume 349 of *Lecture Notes in Computer Science*, pages 375–387. Springer-Verlag, Berlin, 1989.

[12] H.A. Blair and A.L. Brown. Definite clause programs are canonical (over a suitable domain). Technical report, Syracuse University, 1990. To appear on *Annals of Mathematics and Artificial Intelligence*.

[13] A. Bondarenko, F. Toni, and R. A. Kowalski. An assumption-based framework for non-monotonic reasoning. In L. Pereira and A. Nerode, editors, *Logic Programming and Non-monotonic reasoning: Proceedings of the 2nd International Workshop*, pages 171–189, 1993.

[14] P. Bruscoli, F. Levi, G. Levi, and M. C. Meo. Intensional Negation in Constraint Logic Programs. In D. Saccá, editor, *Proc. Eighth Italian Conference on Logic Programming*, pages 359–373, Gizzeria (CZ) Italy, 1993.

[15] L. Cavedon. *On the Completeness of SLDNF-resolution.* PhD thesis, University of Melbourne, Computer Science Department, 1988.

[16] S. Cerrito. Negation and Linear Completion. Presented at the ICLP workshop on Logic Programming Semantics and Non-classical logics, 1991.

[17] D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 111–125. The MIT Press, Cambridge, Mass., 1988.

[18] D. Chan. An Extension of Constructive Negation and its Application in Coroutining. In E. Lusk and R. Overbeck, editors, *Proc. North American Conf. on Logic Programming'89*, pages 477–493. The MIT Press, Cambridge, Mass., 1989.

[19] A. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of the ACM*, 29(3):841–862, 1985.

[20] K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[21] K.L. Clark and F. G. McCabe. *micro-PROLOG:Programming in Logic.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[22] K. L. Clark and S. Gregory. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8:1–49, 1986.

[23] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[24] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un Systeme de Communication Homme-Machine en Francais. Groupe de Reserche en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[25] J. W. de Bakker. Recursive procedures. Tract. N.24, Mathematical Centre, Amsterdam, 1971.

[26] F. S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91, Volume 1 (CAAP'91)*, Lecture Notes in Computer Science, pages 296–319. Springer-Verlag, Berlin, 1991.

[27] Frank S. de Boer and C. Palamidessi. *From Concurrent Logic Programming to Concurrent Constraint Programming.* Lecture notes for the Fourth International School for Computer Science Researchers on *Foundations of Logic Programming*, July 1992, Acireale, Italy.

[28] F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, 1993.

[29] A. Di Pierro. Amalgamating NAI and NAF. In D. Saccá, editor, *Proc. Eighth Italian Conference on Logic Programming*, pages 309–321, Gizzeria (CZ) Italy, 1993.

[30] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation as Instantiation: a new rule for the treatment of negation in Logic Programming. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming.* The MIT Press, Cambridge, Mass., 1991.

[31] A. Di Pierro, M. Martelli, and C. Palamidessi. Sldni-resolution: a non-floundering interpreter for geneneral programs. In P. Asirelli, editor, *Proc. Sixth Italian Conference on Logic Programming.* The MIT Press, Cambridge, Mass., 1991.

[32] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation As Instantiation. Submitted for Publication, 1992.

[33] A. Di Pierro and C. Palamidessi. A Logical Denotational Semantics for Constraint Logic Programming. In *Proc. European Symposium on Programming*, University of Edinburg, Scotland, 1994.

[34] W. Drabent and M. Martelli. Strict Completion of Logic Programs. *New Generation Computing*, 9:69–79, 1991.

[35] W. Drabent. What is Failure? An Approach to Constructive Negation. *Acta Informatica*, 1993. To appear.

[36] Phan Minh Dung. Negation as Hypotheses: an Abductive Foundation for Logic Programming. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*. The MIT Press, Cambridge, Mass., 1991.

[37] K. Eshghi and R. A. Kowalski. Abduction compared with Negation by Failure. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming*, pages 234–254. The MIT Press, Cambridge, Mass., 1989.

[38] M. Falaschi, M. Gabbrielli, K. Marriot, and C. Palamidessi. Compositional analysis for concurrent constraint programming. In *To appear in the Proceedings of the Eight Annual IEEE Symposium on Logic in Computer Science*, 1993.

[39] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[40] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. Technical Report TR 32/89, Dipartimento di Informatica, Università di Pisa, 1989. To appear in *Information and Computation*.

[41] M. Fitting. A Kripke-Kleene semantics for logic programs. In *Journal of Logic Programming*, volume 2, pages 295–312, 1985.

[42] D.M. Gabbay and M. J. Sergot. Negation as inconsistency. *Journal of Logic Programming*, 1:1–35, 1986.

[43] M. Gabbrielli. *The Semantics of Logic Programming as a Programming Language*. PhD thesis, Universitá di Pisa, 1992. Technical Report 17/93.

[44] A. Van Gelder. Negation As Failure Using Tight Derivations for General Logic Programs. *jlp*, 6:109–133, 1989.

[45] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1070–1079. The MIT Press, Cambridge, Mass., 1988.

[46] M. Gelfond and V. Lifschitz. Logic proprams with classical negation. In D. H. D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, 1990.

[47] G. Gierz, K. H. Hoffman, K. Keimel, J. D. Lawson, and D. S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, Berlin, Heidelberg New York, 1980.

[48] J. Y. Girard. *Proof Theory and Logical Complessity*, 1987. Bibliopolis, Napoli.

[49] W.G. Golson. Toward a declarative semantics for infinite objects in logic programming. *Journal of Logic Programming*, 5:151–164, 1988.

[50] D. De Groot and G. Lindstrom eds. *Logic Programming, Functions, Relations and Equations*. Prentice-Hall, 1986.

[51] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I)*. North-Holland, 1971.

[52] L. J. Henschen and H. S. Park. Compiling the GCWA in Indefinite Databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 395–438. Morgan Kaufmann, Los Altos, Ca., 1988.

[53] J. Jaffar, , and P.J. Stuckey. Canonical logic programs. *Journal of Logic Programming*, 3:143–155, 1986.

[54] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[55] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Department of Computer Science, Monash University, June 1986.

[56] J. Jaffar, J.-L. Lassez, and J. W. Lloyd. Completeness of the Negation as Failure Rule. In *Proc. IJCAI-83*, pages 500–506, Karlsruhe, 1983.

[57] J. Jaffar, J. L. Lassez, and J. Maher. A theory of complete logic programs with equality. *Journal of Logic Programming*, 3:211–223, 1984.

[58] J. Jaffar, J. L. Lassez, and J. Maher. Some issues and trends in the semantics of logic programming. In K. A. Bowen and E. Shapiro, editors, *Proc. 3rd Int. Conf. on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 223–241. Springer-Verlag, 1986.

[59] R. Jagadeesan, V.A. Saraswat, and V. Shanbhogue. Angelic non-determinism in concurrent constraint programming. Technical report, Xerox Park, 1991.

[60] P. T. Johnstone. *Stone Spaces.* Cambridge University Press, Cambridge, 1982.

[61] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proc. of IFIP Congress*, pages 471–475, New York, 1974. North-Holland.

[62] A. C. Kakas and P. Mancarella. Negation as Stable Hypothesis. In A. Nerode, W. Marek, and V. S. Subramahnian, editors, *Logic Programming and Non-monotonic reasoning: Proceedings of First International Workshop*, pages 275–288. The MIT Press, 1991.

[63] S. C. Kleene. *Introduction to Metamathematics.* Van Nostrand, Princeton, 1952.

[64] R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[65] R. A. Kowalski. Predicate Logic as a Programming Language. In *Information Processing 74*, pages 569–574. North-Holland, 1974.

[66] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[67] K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7(3):231–245, 1989.

[68] M. Z. Kwiatkowska. Infinite Behaviour and Fairness in Concurrent Constraint Programming. In J. W. de Bakker, W. P.de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 348–383, Beekbergen The Nederland, June 1992. REX Workshop, Springer-Verlag, Berlin.

[69] J.-L. Lassez and M. J. Maher. Closures and Fairness in the Semantics of Programming Logic. *Theoretical Computer Science*, 29:167–184, 1984.

[70] G. Levi, M. Martelli, and C. Palamidessi. Failure and success made symmetric. In S. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 3–22. The MIT Press, Cambridge, Mass., 1990.

[71] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

[72] J. W. Lloyd and R. W. Topor. Making Prolog more Expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[73] J. Maluszyński and T. Näslund. Fail Substitutions for Negation as Failure. In E. Lusk and R. Overbeck, editors, *Proc. North American Conf. on Logic Programming'89*, pages 461–476. The MIT Press, Cambridge, Mass., 1989.

[74] P. Martin-Löf. *Intuitionistic Type Theory*, 1984. Bibliopolis, Napoli.

[75] P. Martin-Löf. Mathematics of infinity. In P. Martin-Löf and G. E. Mints, editors, *COLOG-88 Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 147–196. Springer-Verlag, Berlin, 1989.

[76] J. Minker. On indefinite database and the closed world assumption. In *Sixth Conference on Automated Deduction*, pages 292–308. Springer-Verlag, Berlin, 1982.

[77] M.A. NaitAbdallah. On the intepretation of infinite computations in logic programming. In J. Paredaens, editor, *Proc. of Automata, Languages and Programming*, volume 172, pages 374–381. Springer Verlag, 1984.

[78] S. O. Nyström and B. Jonsson. Indeterminate Concurrent Constraint Programming: A Fixpoint Semantics for Non-Terminating Computations. In D. Miller, editor, *Proc. of the 1993 International Logic Programming Symposium*, Series on Logic Programming, pages 335–352. The MIT Press, 1993.

[79] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. of the 5th GI conference*, Lecture Notes in Computer Science, pages 167–183. Springer-Verlag, 1981.

[80] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[81] G. Plotkin. Domains. Department of Computer Science, University of Edimburgh, 1992. Post-graduate lecture notes in advanced domain theory (incorporating the 'Pisa notes' 1981).

[82] G. D. Plotkin. A powerdomain construction. *SIAM*, 5(3), 1976.

[83] D. L. Poole and R. Goebel. Gracefully adding negation and disjunction to prolog. In E. Y. Shapiro, editor, *Proc. Third Int'l Conf. on Logic Programming*, pages 635–641. The MIT Press, Cambridge, Mass., 1986.

[84] T. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, Ca., 1988.

[85] T. Przymusinski. On Constructive Negation in Logic Programming. Draft paper at NACLP'89, 1989.

[86] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[87] K. Sakai and T. Miyachi. Incorporating Naive Negation into Prolog. ICOT Pre-publication Report, 1985.

[88] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. Seventeenth Annual ACM Symp. on Principles of Programming Languages.* ACM, 1990.

[89] V. A. Saraswat, M. Rinard, and P. Panangadnen. Semantic foundation of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages.* ACM, 1991.

[90] D. Scott. Domains for Denotational Semantics. In *Proc. ICALP*, volume 140 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1982.

[91] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Survey*, 21(3), 1989.

[92] E. Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, Tokyo, 1983.

[93] J. C. Shepherdson. Negation as failure: a comparison of Clark's completed database and Reiter's closed world assumption. *Journal of Logic Programming*, 1(1):51–79, 1984.

[94] J. C. Shepherdson. Negation as Failure II. *Journal of Logic Programming*, 2(3):185–202, 1985.

[95] J. C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1988.

[96] J. C. Shepherdson. A sound and complete semantics for a version of negation as failure. *Theoretical Computer Science*, 65:343–371, 1989.

[97] M. B. Smyth. Power domains. *JCSS*, 16(1), 1978.

[98] M. B. Smyth. Powerdomains and predicate transformers: a topological view. In J. Diaz, editor, *Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 662–675. Springer-Verlag, Berlin, 1983.

[99] R. F. Stärk. A complete axiomatization of the three-valued completion of logic programs. *Journal of Logic and Computation*, 1(6):811–834, 1991.

[100] R. F. Stärk. *The Proof Theory of Logic Programs with Negation.* PhD thesis, University of Berne, 1992.

[101] J. E. Stoy. *The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1979.

[102] P. Stuckey. Constructive negation for constraint logic programming. In *Proc. sixth Annual Symposium on Logic in Computer Science*, 1991.

[103] D. Turi. Logic Programs with Negation: Classes, Models and Interpreters. Technical Report CS-R8943, Centre for Mathematics and Computer Science, 1989.

[104] M. H. van Emden and G. J. de Lucena. Predicate logic as language for parallel programming. In K. L. Clark and S. A. Tä rnlund, editors, *Logic Programming*, pages 189–198. Academic Press, London, 1982.

[105] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[106] A. van Gelder, K. Ross, and J. S. Schlipf. Unfounded Sets and Well-Founded Semantics for General Programs. In *Proc. ACM Symp. on Principles of Database Systems*, pages 221–230. ACM, 1988.

[107] A. van Gelder and J. S. Schlipf. *Common Sense Axiomatizations for Logic Programming*, 1993. To appear in Journal of Logic Programming.

[108] J. Vauzeilles. Negation as Failure and Intuitionistic Three-Valued Logics. Presented at the ICLP workshop on Logic Programming Semantics and Non-classical logics, 1991.

[109] S. Vickers. *Topology via Logic.* Cambridge University Press, 1990.

[110] D. Wolfram, J. L. Lassez, and M.Maher. A unified treatment of resolution strategies for logic programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 263–276, 1984.

[111] Guo-Qiang Zhang. *Logic of Domains.* Birkhauser Boston, 1991.