An algebraic perspective of constraint logic programming*

Frank S. de Boer† and Alessandra Di Pierro‡ and Catuscia Palamidessi§

Abstract

We develop a denotational, fully abstract semantics for constraint logic programming (clp) with respect to successful and failed observables. The denotational approach turns out very useful for the definition of new operators on the language as the counterpart of some abstract operations on the denotational domain. In particular, by defining our domain as a *cylindric Heyting algebra*, we can exploit, to this aim, operations of both cylindric algebras (such as cylindrification), and Heyting algebras (such as implication and negation). The former allows us to generalize the clp language by introducing an explicit hiding operator; the latter allows us to define a notion of negation which extends the classical negation used in Logic Programming. In particular, we show that our notion subsumes both Negation as Failure ([7]) and Negation as Instantiation ([10]).

1 Introduction

Constraint logic programming (clp, [18]) is an extension of logic programming ([27]) in which the concept of unification on the Herbrand universe is replaced by the more general notion of constraint over an arbitrary domain. A program is a set of clauses possibly containing some constraints. A computation consists of a sequence of goals with constraints, where each goal is obtained from the previous one by replacing an atom by the body of a defining clause, and by adding the corresponding constraint, provided that consistency is preserved.

Like pure logic programming, clp has a natural computational model based on the so-called *process interpretation* ([26, 22]): the conjunction of atoms in a goal can be regarded as parallelism, and the selection of alternative clauses as nondeterminism. Such a model presents many similarities with the paradigm of concurrent constraint programming (ccp, [25]). However, it differs from the latter (at least, from the version in [25]) because it supports the notion of consistency: an action can be performed only if it does not lead to an inconsistent store. On the other hand, ccp is provided with a special primitive for synchronization, which clp does not have.

The aim of this work is to reformulate clp and its semantics in algebraic terms, thus obtaining a *calculus of agents*, communicating via a common store

*This work has been supported by the HCM project EXPRESS.

†Faculteit Wiskunde en Informatica, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. E.mail: frankb@cs.ruu.nl

‡Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. E.mail: dipierro@di.unipi.it

§DISI, Università di Genova, via Benedetto XV 3, 16132 Genova, Italy. E.mail: catuscia@disi.unige.it

of constraints. Our language, which we call Γ_clp, is an extension of clp, in the sense that goals are defined as the free term algebra generated by constraints, disjunction, conjunction and existential quantification. This approach has the advantage that we can define the operational semantics in a structured way, following the so-called SOS method ([20]), via a transition system where the transition relation expresses the computation step. Thus the behavior of new operators can in principle be defined simply by adding new rules.

The denotational semantics of Γ_clp is defined via a homomorfism into a *cylindric Heyting algebra* whose elements are downward-closed sets of constraints. One of the advantages of the denotational approach is that it allows to define possible extensions of the language by reasoning at the abstract level, i.e. by looking for possible operators which are definable on the domain of denotation. We give an example of this method by considering intuitionistic negation, i.e. a form of negation for which the law of excluded middle, $\phi \vee \neg\phi$, does not hold. Further we show that these operations can be defined also operationally by enriching the transition system with suitable rules.

Unexpectedly, this form of negation happens to be an extension of the well-known concept of *negation as failure*, which is defined operationally in clp by an inference rule of the form

$$\frac{\text{all fair computations of } \;\text{:-}\; A \text{ fail}}{\neg A}.$$

for an atom A. Analogously, we can capture the extension of negation as failure, proposed by Shepherdson ([23]), which allows to infer that :- $\neg A$ succeeds with answer substitution θ whenever the goal :- $A\theta$ fails.

Also the notion of *negation as instantiation* ([10]) can be expressed in terms of our intuitionistic negation. This notion is defined in [10] by means of the inference rule

$$\frac{\text{all fair computations of } \;\text{:-}\; A \text{ finitely instantiate some variables of } A}{\exists\neg A}.$$

From the point of view of the underlying transition system, the above two rules are actually *meta-rules*. In fact the premises speak about the non-existence of successful derivations, i.e. sequences of transitions. The successful computation of a negative agent, as formalized by these rules, is done in one step.

Our characterization of negation, on the contrary, defines the stepwise evolution of a negative agent. Our rule is a "real transition rule", in the SOS sense, and does not contain negative premises. The resulting operational semantics actually is an *inductive operational semantics* ([4]), i.e. defined by an *inductive system* ([3]), with the advantages that this approach implies (see for instance [4] and [5]). Moreover, we show that our treatment of negation allows a natural declarative interpretation of the operational semantics.

2 Constraint System

The concept of constraint is central for the paradigm of clp and represents its major novelty with respect to logic programming. We follow here the approach of [24], which defines the notion of constraint system along the lines of Scott's *information systems* ([21]). Intuitively, an information system consists of a set of elements each of them represents some "consistent information", together with an *entailment relation* $\vdash$ which establishes which elements can be derived from which other ones. In the view of [24], a constraint system is the same kind of structure, but for the presence of an additional element representing inconsistency. The term "constraint" refers to the fact that the information usually involve variables, i.e. they establish bounds to the range of values that such variables can assume.

The advantage of this approach, with respect to the notion of constraint system as a first order logic theory used in [18], is that it is more abstract, i.e. less relations among constraints are required to hold. For instance, all the axioms for the existential quantifier listed below are valid in the First Order Predicate Calculus, but not all the properties of the first order existential quantifier are necessarily valid here. Therefore this approach is suitable to understand the abstract features that the notion of constraint must have, in order to define the semantics of clp. Of course, when using clp in practice, it is usually better to deal with a constraint system á la [18]. The results we obtain in this paper, however, would still be valid in that framework.

Following [24], we regard a constraint system as a lattice provided with a top and a bottom element[1]. However, we consider the ordering relation $\sqsubseteq$ as representing the entailment relation instead of its reverse as it is done in [24]. Thus, by $c \sqsubseteq d$ we mean that c "entails" d or c "is logically stronger than" d. The reason of this choice is that we want to maintain the meaning assigned to the ordering relation in the "algebraic view" of logic ([28]). Therefore, in our constraint systems the top element is *true*, representing the element implied by (with less information than) anything; the bottom element is *false*, representing inconsistency, i.e. the element logically stronger (with more information) than anything; the *greatest lower bound* (glb), which we shall denote by $\sqcap$, represents the conjunction of information and corresponds to the logical *and*. The reverse operation, i.e. the *least upper bound* (lub), will be represented by $\sqcup$. As we will see, the latter does not correspond to the logical *or*.

Definition 1 A constraint system is a lattice $(\mathcal{C}, \sqsubseteq, \sqcup, \sqcap, \mathit{true}, \mathit{false})$ where $\sqcup$ is the lub operation, $\sqcap$ is the glb operation, and *true*, *false* are the greatest and least elements of $\mathcal{C}$, respectively.

2.1 Cylindric Constraint Systems

In order to model local variables in Γ_clp, we need a sort of hiding operator. This can be formalized by introducing a notion of constraint system which supports *cylindrification operators*, a concept borrowed from the theory of cylindric algebras, by Henkin, Monk and Tarski ([17]).

Furthermore, in order to model parameter passing, it will be useful to enrich the constraint system with the so-called *diagonal constraints*, based on the notion of diagonal elements, also from [17].

Assume given a (denumerable) set of variables *Var* with typical elements $x, y, z, \ldots$, and consider a family of operators $\{\exists_x \mid x \in \mathit{Var}\}$ (cylindrification operators) and of constants $\{\delta_{xy} \mid x, y \in \mathit{Var}\}$ (diagonal elements). Starting from a constraint system $\mathcal{C}$, define a cylindric constraint system as the smallest set $\mathcal{C}'$ such that

$$\mathcal{C}' = \mathcal{C} \cup \{\exists_x c \mid x \in \mathit{Var},\ c \in \mathcal{C}'\} \cup \{\delta_{xy} \mid x, y \in \mathit{Var}\}$$

modulo the identities and with the additional relations derived by the following axioms from [17]:

[1] Actually, in [24] a constraint system is required to be a *complete and algebraic* lattice, in order to deal with infinite computations. Here we relax these requirements because for the moment we consider only finite computations. In Sections 8 and 9 we will consider also the limit results of infinite computations.

(i) $c \sqsubseteq \exists_x c$,

(ii) if $c \sqsubseteq d$ then $\exists_x c \sqsubseteq \exists_x d$,

(iii) $\exists_x (c \sqcap \exists_x d) = \exists_x c \sqcap \exists_x d$,

(iv) $\exists_x \exists_y c = \exists_y \exists_x c$,

(v) $\delta_{xx} = true$,

(vi) if $z \neq x, y$ then $\delta_{xy} = \exists_z (\delta_{xz} \sqcap \delta_{zy})$,

(vii) if $x \neq y$ then $\delta_{xy} \sqcap \exists_x (c \sqcap \delta_{xy}) \sqsubseteq c$.

Note that these laws are satisfied if $\exists_x$ is taken to be the first-order *existential operator* and δ_{xy} is taken to be the equality between x and y. However these axioms are weaker than first-order logic axioms and do not identify uniquely the families of $\exists_x$ and δ_{xy} operators. We are not concened with defining them uniquely, anyway. Rather, these axioms are designed for the purpose of this paper, which is to study the semantics of clp. The results obtained in this paper make no other assumptions, about the underlying constraint system, than those specified by the laws above, and therefore they are valid for all those instances of clp in which there is a notion of *existential operator* and equality which satisfy them.

The combined use of the cylindrification operators and diagonal elements allow us to model variable renaming by representing $\phi[y/x]$ as the formula $\exists_x (\delta_{xy} \sqcap \phi)$. In fact, if cylindrification is interpreted as the first-order existential operator, and δ_{xy} as equality between x and y, then $\exists_x (\delta_{xy} \sqcap \phi)$ has precisely the meaning of the formula derived from ϕ by replacing all the free occurrences of x by y.

Note that a cylindric constraint system is not necessarily a cylindric algebra (see Definition 1.1.1 of [17]). In fact, as we will see in Section 5.1, the structure $(\mathcal{C}, \sqcup, \sqcap, \mathit{true}, \mathit{false})$ is in general not distributive, hence it is not a Boolean algebra.

From (i) and (iii) it follows that $\exists_x$ is idempotent, i.e., for all $c \in \mathcal{C}$, $\exists_x c = \exists_x \exists_x c$. This means that it is also a kernel operator ([16]).

3 The language clp

The syntax and the computational mechanism of clp are very simple. 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 constraints such that $c_1 \sqcap \ldots \sqcap c_m$ is consistent, and the B_i's are *atoms*, i.e. predicates applied to terms. The intended interpretation of the comma symbol "," is logical conjunction.

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 constraints such that $d_1 \sqcap \ldots \sqcap d_n$ 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 with predicate, say, p, by the body of a clause defining p, nondeterministically selected among those clauses whose constraint is consistent with the constraint of the goal. A clause is renamed each time it is used, so to avoid variable clashes with the goal. More formally, given a renamed clause like the above one, a *computation step* rewrites a goal

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

into

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

provided that the constraint $(B_i = A) \sqcap c_1 \sqcap \ldots c_m \sqcap d_1 \sqcap \ldots \sqcap d_n$ is consistent[2], 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 $c_1 \sqcap \ldots \sqcap c_n$.

4 The language Γ_clp

The syntax and the computational mechanism of clp are very simple but they do not provide a suitable basis for a structured operational semantics and for a denotational semantics. We need to reformulate the syntax of clauses and goals by means of a free grammar. The resulting language will be called Γ_clp.

Constraints and atoms are basic constructs also in our language[3]. 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 symbol :- in the clauses corresponds to double implication. The $\exists_x$ symbol here must not be confused with the analogous operator of the constraint system, but, as we will see later, there is a close correspondence between them.

The grammar is described in Table 1. The language is parametric with respect to $\mathcal{C}$, and so is the semantic construction developed in this paper. We will assume in the following that there is at most one declaration for each predicate.

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. 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

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

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

5 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

[2]The underlying constraint system is assumed to embed the equality theory, and $p(t_1, \ldots, t_m) = q(u_1, \ldots, u_n)$ stands for $t_1 = u_1 \sqcap \ldots \sqcap t_m = u_n$ if $p = q$ and $m = n$, *false* otherwise.

[3]We restrict, without loss of generality, to atoms of the form $p(x)$.

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

Table 1: The language Γ_clp. The symbol p ranges over predicate names, and c ranges over the elements of a cylindric constraint system $\mathcal{C}$.

as an agent, a conjunction of atoms as parallel agents which communicate with each other by establishing constraints on the global variables, and a disjunction we regard as a choice. 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 ([20]), 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.

5.1 Constraint associated to a goal

In order to avoid useless computations, in clp each resolution step is subjected to a consistency check. In other words, a computation is aborted as soon as an inconsistent situation is detected. This check is done, of course, on the information which is *present* in the goal, which we call *constraint associated* to the goal. Such notion must not be confused with the *constraint generated* by a goal, which is the *future*, final information that the goal will have accumulated at the end of the computation(s).

In order to mimic the consistency check in Γ-clp we have first of all to clarify what is the constraint associated to a goal. In particular, we have to define what is the constraint associated to a $\vee$-goal, and how does it combine with the constraint of a parallel agent. 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*. A first idea would be to define the constraint associated to a disjunction as the lub $\sqcup$ of the two constraints of the disjuncts. Unfortunately this choice does not work when the constraint system is not distributive. Consider for example the constraint system illustrated in Figure 1. If we have the goal $x = 0 \vee x = 1$, then the associated constraint would be $x = 0 \sqcup x = 1$, namely *true*, which is consistent with the constraint $x = 2$. On the other hand, the goal $(x = 0 \vee x = 1) \wedge x = 2$ should fail, given the interpretation of $\vee$ as logical or.

Another example is the following: consider the constraint system illustrated in Figure 2, and consider the goal $x < 0 \vee x > 0$ in parallel with the goal $x \geq 0$. The constraint associated with the first goal would be $x < 0 \sqcup x > 0$, equivalent to *true*, which, together with $x \geq 0$, gives $x \geq 0$. On the other hand, the

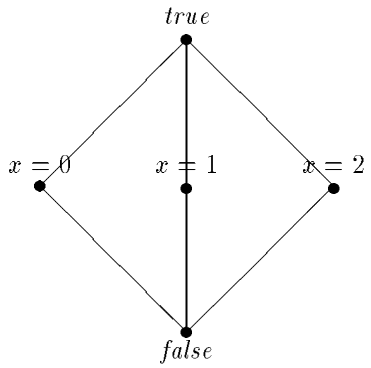

Figure 1: A non-distributive constraint system.

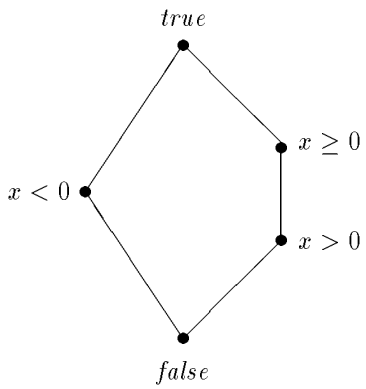

Figure 2: A non-distributive constraint system.

computation of $(x < 0 \vee x > 0) \wedge x \geq 0$ should intuitively discard $x < 0 \wedge x \geq 0$ as inconsistent and deliver $x > 0 \wedge x \geq 0$, i.e. $x > 0$.

The problem is that, in order to correspond to the logical "and" and "or", $\sqcap$ and $\sqcup$ 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 these examples: the lattices in Figures 1 and 2 are not distributive.

A possible solution is to embed the constraint system into a distributive one, where the lub and the glb model logical disjunction and conjunction. A simple way to do this is to lift to sets of constraints, following the idea of [9]. In fact, set union and set intersection, which are respectively the lub and the glb on sets, enjoy the distributive property.

We consider only sets which are *downward-closed*. This choice corresponds to the intention of modeling the operational counterpart of the set of d's which satisfy $D \models d \Rightarrow G$, for a given program D and goal G (we will come back to this declarative interpretation in the sections 7 and 9).

We recall that the downward closure of a set $C \subseteq \mathcal{C}$ is the set $\{c \in \mathcal{C} \mid$ there exists $d \in C$ such that $c \sqsubseteq d\}$, which we denote by $\downarrow C$. A set $C \subseteq \mathcal{C}$ is downward-closed iff $C = \downarrow C$. We will denote by $\mathcal{P}_{\mathrm{d}}(\mathcal{C})$ the set of non-empty, downward-closed subsets of $\mathcal{C}$. In the following, we call a set $C \in \mathcal{P}(\mathcal{C})$ *finitary* iff there exist a *finite* set of constraints $\{c_i \mid i \in I\}$ such that $C = \bigcup_{i \in I} \downarrow c_i$.

Given a cylindric constraint system $(\mathcal{C}, \sqsubseteq, \sqcup, \mathit{true}, \mathit{false}, \mathit{Var}, \exists, \delta)$, consider the structure

$$\mathbf{C} = (\mathcal{P}_{\mathrm{d}}(\mathcal{C}), \subseteq, \cup, \cap, \mathit{False}, \mathit{True}, \mathit{Var}, \exists\!\!\exists, \Delta),$$

where *False* is $\downarrow \mathit{false}$, which coincides with $\{\mathit{false}\}$, and *True* is $\downarrow \mathit{true}$, which coincides with $\mathcal{C}$. For $C \in \mathcal{P}_{\mathrm{d}}(\mathcal{C})$, $\exists\!\!\exists_x C$ is defined as $\downarrow \exists_x C$, where $\exists_x C$ stands for the pointwise application of $\exists_x$ to the elements of C. Finally, Δ_{xy} is defined as $\downarrow \delta_{xy}$. The embedding of the original constraint system into $\mathbf{C}$ is obtained by mapping each element c into $\downarrow c$. We have the following property:

Lemma 2 The embedding of the original constraint system into $\mathbf{C}$ preserves the ordering, the glb and the existential operator. Namely, for $c, d \in \mathcal{C}$ we have:

(1) $c \sqsubseteq d \Leftrightarrow \downarrow c \subseteq \downarrow d$;

(2) $\downarrow (c \sqcap d) = \downarrow c \cap \downarrow d$;

(3) $\downarrow \exists_x c = \exists\!\!\exists_x \downarrow c$.

Proof (1) and (2) are obvious. For (3) we have:

($\subseteq$) Assume $d \in \downarrow \exists_x c$. Then $d \sqsubseteq \exists_x c$. By Axiom (ii) and idempotency of $\exists_x$, $\exists_x d \sqsubseteq \exists_x c$ holds. Then, for $c' = c \sqcap \exists_x d$, we have that $c' \in \downarrow c$, and, by Axiom (iii), $\exists_x c' = \exists_x c \sqcap \exists_x d = \exists_x d$, i.e. $d \in \exists\!\!\exists_x \downarrow c$.

($\supseteq$) Assume $d \in \exists\!\!\exists_x \downarrow c$. Then there exists $c' \sqsubseteq c$ such that $d \sqsubseteq \exists_x c'$. By Axiom (ii) we have $\exists_x c' \sqsubseteq \exists_x c$, hence we derive $d \sqsubseteq \exists_x c$, i.e. $d \in \downarrow \exists_x c$.

Note that the analogous of Lemma 2 (2) for $\sqcup$ and $\cup$ does not hold. The lub $\cup$ in $\mathbf{C}$ satisfies more properties. In fact, since the glb and the lub are set-intersection and set-union respectively, we have:

Remark 3 The support lattice of $\mathbf{C}$ is complete and distributive.

We show now that $\mathbf{C}$ is a cylindric constraint system:

Proposition 4 $\mathbf{C}$ is a cylindric constraint system ordered by set-inclusion.

Proof It is sufficient to show that $\exists\!\!\exists$ and Δ satisfy Axioms (i)-(vii) of Section 2.1. We use the following lemma:

Lemma 5 For every $x \in \mathit{Var}$, $\exists\!\!\exists_x$ is idempotent, namely $\exists\!\!\exists_x \exists\!\!\exists_x C = \exists\!\!\exists_x C$ for all $C \in \mathcal{P}_{\mathrm{d}}(\mathcal{C})$.

Proof $(\supseteq)$

$$\begin{array}{ll} & \exists\!\!\exists_x C \\ = & \\ & \downarrow \exists_x C \\ = & \{\text{idempotency of } \exists_x\} \\ & \downarrow \exists_x \exists_x C \\ \subseteq & \\ & \downarrow \exists_x \downarrow \exists_x C \\ = & \\ & \exists\!\!\exists_x \exists\!\!\exists_x C. \end{array}$$

$(\subseteq)$

$$\begin{array}{ll} & \exists\!\!\exists_x \exists\!\!\exists_x C \\ = & \\ & \downarrow \exists_x \downarrow \exists_x C \\ \subseteq & \{\text{monotonicity of } \exists_x\} \\ & \downarrow \exists_x \exists_x C \\ = & \{\text{idempotency of } \exists_x\} \\ & \downarrow \exists_x C \\ = & \\ & \exists\!\!\exists_x C. \end{array}$$

We prove now Proposition 4. In the following, C, C' are elements of $\mathcal{P}_{\mathrm{d}}(\mathcal{C})$ and x, y, z are elements of Var.

(i) $C \subseteq \exists\!\!\exists_x C$. In fact, for each $c \in C$, $c \sqsubseteq \exists_x c$, hence $c \in \downarrow \exists_x c \subseteq \downarrow \exists_x C = \exists\!\!\exists_x C$.

(ii) $C \subseteq C'$ implies $\exists\!\!\exists_x C \subseteq \exists\!\!\exists_x C'$. In fact, $C \subseteq C' \Rightarrow \exists_x C \subseteq \exists_x C' \Rightarrow \downarrow \exists_x C \subseteq \downarrow \exists_x C'$.

(iii) $\exists\!\!\exists_x (C \cap \exists\!\!\exists_x C') = \exists\!\!\exists_x C \cap \exists\!\!\exists_x C'$. In fact, for the left-to-right set-inclusion we have: $\exists\!\!\exists_x (C \cap \exists\!\!\exists_x C') \subseteq \exists\!\!\exists_x C$, by monotonicity of $\exists\!\!\exists_x$ (i.e. Axiom (ii)), and $\exists\!\!\exists_x (C \cap \exists\!\!\exists_x C') \subseteq \exists\!\!\exists_x \exists\!\!\exists_x C' = \exists\!\!\exists_x C'$, by monotonicity and idempotency of $\exists\!\!\exists_x$ (Lemma 5). Hence we derive $\exists\!\!\exists_x (C \cap \exists\!\!\exists_x C') \subseteq \exists\!\!\exists_x C \cap \exists\!\!\exists_x C'$. For the other inclusion, let $d \in \exists\!\!\exists_x C \cap \exists\!\!\exists_x C'$. Then there exist $c \in C, c' \in C'$ such that $d \in \downarrow \exists_x c \cap \downarrow \exists_x c'$. Hence, by Lemma 2 (2), $d \in \downarrow (\exists_x c \sqcap \exists_x c')$. By Axiom (iii) on $\mathcal{C}$, we have $d \in \downarrow \exists_x (c \sqcap \exists_x c')$. Finally observe that, since C is downward closed, $c \sqcap \exists_x c' \in C \cap \downarrow \exists_x C' = C \cap \exists\!\!\exists_x C'$.

(iv) $\exists\!\!\exists_x \exists\!\!\exists_y C = \exists\!\!\exists_y \exists\!\!\exists_x C$. In fact,

$$\begin{array}{ll} & \exists\!\!\exists_x \exists\!\!\exists_y C \\ = & \\ & \downarrow \exists_x \downarrow \exists_y C \\ = & \{\text{monotonicity of } \exists_x\} \\ & \downarrow \exists_x \exists_y C \\ = & \{\text{Axiom (iv) on } \mathcal{C}\} \\ & \downarrow \exists_y \exists_x C \\ = & \{\text{monotonicity of } \exists_y\} \\ & \downarrow \exists_y \downarrow \exists_x C \\ = & \\ & \exists\!\!\exists_y \exists\!\!\exists_x C. \end{array}$$

(v) $\Delta_{xx} = \mathit{True}$. In fact, $\Delta_{xx} = \downarrow \delta_{xx}$, and, by Axiom (v) on $\mathcal{C}$, $\delta_{xx} = \mathit{true}$.

(vi) If $z \neq x, y$, then $\Delta_{xy} = \exists\!\!\exists_z(\Delta_{xz} \cap \Delta_{zy})$. In fact,

$$\begin{array}{ll} & \Delta_{xy} \\ = & \\ & \downarrow \delta_{xy} \\ = & \{\text{Axiom (vi) on } \mathcal{C}, \text{ under the assumption } z \neq x, y\} \\ & \downarrow \exists_z(\delta_{xz} \sqcap \delta_{zy}) \\ = & \{\text{monotonicity of } \exists_z\} \\ & \downarrow \exists_z \downarrow (\delta_{xz} \sqcap \delta_{zy}) \\ = & \{\text{Lemma 2 (2)}\} \\ & \downarrow \exists_z(\downarrow \delta_{xz} \cap \downarrow \delta_{zy}) \\ = & \\ & \exists\!\!\exists_z(\Delta_{xz} \cap \Delta_{zy}). \end{array}$$

(vii) If $x \neq y$, then $\Delta_{xy} \cap \exists\!\!\exists_x(C \cap \Delta_{xy}) \subseteq C$. In fact, for $c \in \Delta_{xy} \cap \exists\!\!\exists_x(C \cap \Delta_{xy})$, we have $c \in \downarrow \delta_{xy}$ and $c \in \downarrow \exists_x(C \cap \downarrow \delta_{xy})$. Hence there exists $c' \in C \cap \downarrow \delta_{xy}$ such that $c \sqsubseteq \exists_x c'$. Since $c' \sqsubseteq \delta_{xy}$, we obtain $c \sqsubseteq \exists_x(c' \sqcap \delta_{xy})$. Therefore we have $c \sqsubseteq \delta_{xy} \sqcap \exists_x(c' \sqcap \delta_{xy})$. By using Axiom (vii) on $\mathcal{C}$, under the assumption $x \neq y$, we derive $c \sqsubseteq c'$, and we conclude $c \in C$ because C is downward-closed.

The operator $\exists\!\!\exists$ enjoys another important property, namely linearity (and therefore continuity):

Proposition 6 $\exists\!\!\exists_x$ is linear on $\mathcal{C}$, i.e. $\exists\!\!\exists_x \bigcup_{i\in I} S_i = \bigcup_{i\in I} \exists\!\!\exists_x S_i$, for all families $\{S_i\}_{i\in I} \subseteq \mathcal{P}_{\mathrm{d}}(\mathcal{C})$.

Proof

$$d \in \exists\!\!\exists_x \bigcup_{i\in I} S_i$$

$\Leftrightarrow$

there exists $c \in \bigcup_{i\in I} S_i$ such that $d \sqsubseteq \exists_x c$

$\Leftrightarrow$

for some $i \in I$ there exists $c \in S_i$ such that $d \sqsubseteq \exists_x c$

$\Leftrightarrow$

for some $i \in I$, $d \in \exists\!\!\exists_x S_i$

$\Leftrightarrow$

$$d \in \bigcup_{i\in I} \exists\!\!\exists_x S_i.$$

The entailment relation on $\mathbf{C}$, i.e. the operational counterpart of $\subseteq$, 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 ($\exists\!\!\exists_x$ and Δ_{xy}) corresponding to the laws of Proposition 4.

Now we can define the function $con : Goals \to \mathbf{C}$, which gives the constraint established by a goal.

Definition 7

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

Note that the conjunction in $\mathbf{C}$ is represented by set intersection, and it corresponds to classical conjunction. The disjunction, on the other hand, is represented by set union and it *does not* correspond to classical disjunction, but rather to intuitionistic disjunction. In fact, if we consider for instance the constraint system in Figure 2, we have $con(x < 0 \vee x \geq 0) = con(x < 0) \cup con(x \geq 0) = (\downarrow x < 0) \cup (\downarrow x \geq 0) \neq True$.

The consistency check on a goal G consists in verifying that $con(G)$ is consistent.

5.2 The problem of the consistency check

To prove the consistency of C means to prove that $C \neq False$, or, equivalently, that $C \nVdash False$.

To this purpose, having a complete deduction system for $\Vdash$ is not sufficient, because it does not imply the computability of $\nVdash$. Fortunately, we only need to prove consistency of constraints associated to finite goals, and it is easy to see that they are finitely generated by the following grammar:

$$C ::= \downarrow c \mid C \cap C \mid \exists\!\!\exists_x C \mid C \cup C.$$

We call $\mathcal{L}$ the language generated by this grammar. For these constraints we have a complete system to infer the satisfiability, which is described in Table 2.

(1) $\dfrac{}{\downarrow c = \downarrow c}$	(2) $\dfrac{C_1 \;=\; \cup_i \downarrow c_i \;\;,\;\; C_2 \;=\; \cup_j \downarrow d_j}{C_1 \cap C_2 \;=\; \cup_{ij} \downarrow (c_i \sqcap d_j)}$
(3) $\dfrac{C \;=\; \cup_i \downarrow c_i}{\exists\!\!\exists_x C \;=\; \cup_i \downarrow \exists_x c_i}$	(4) $\dfrac{C_1 \;=\; \cup_i \downarrow c_i \;\;,\;\; C_2 \;=\; \cup_j \downarrow d_j}{C_1 \cup C_2 \;=\; (\cup_i \downarrow c_i) \cup (\cup_j \downarrow d_j)}$
(5) $\dfrac{}{sat(\downarrow c)} \quad c \neq \mathit{false}$	(6) $\dfrac{C \;=\; \cup_i \downarrow c_i \;\;,\;\; \exists j : \; sat(\downarrow c_j)}{sat(C)}$

Table 2: Deduction system for the consistency check. sat stands for "satisfiable", i.e. consistent.

The idea behind this system is to reduce the expressions to disjunctions of elementary expressions of the form $\downarrow c$, whose consistency we are able to test directly on the underlying constraint system. In fact, the following holds:

Remark 8 Let $c \in \mathcal{C}$. Then $\downarrow c \neq \downarrow \mathit{false}$ iff $c \neq \mathit{false}$.

Observe that the assumption that $c \neq \mathit{false}$ is semidecidable (hence decidable) is customary for the constraint systems used in clp.

Relative completeness of the deduction system in Table 2, where the term "relative" means that completeness depends upon the assumption that we are able to prove $c \neq \mathit{false}$ whenever this holds, follows from the following lemma:

Lemma 9 The first four rules of Table 2 imply that for each element $C \in \mathcal{L}$, C is finitary.

Proof By induction on the structure of C:

($C = \downarrow c$) Obvious, using Rule (1).

($C = C_1 \cap C_2$) By induction, there exist finite sets $\{c_i \in \mathcal{C} \mid i \in I\}$, $\{d_j \in \mathcal{C} \mid j \in J\}$ such that $C_1 = \bigcup_{i \in I} \downarrow c_i$ and $C_2 = \bigcup_{j \in J} \downarrow d_j$ hold, and these equalities can be derived. By distributivity of set-intersection with respect to set-union, we then have $C_1 \cap C_2 = \bigcup_{i \in I, j \in J} (\downarrow c_i \cap \downarrow d_j)$.By Lemma 2 (2) we obtain $C_1 \cap C_2 = \bigcup_{i \in I, j \in J} \downarrow (c_i \sqcap d_j)$. This equality can be inferred by Rule (2).

($C = \exists\!\!\exists_x C'$) By induction, there exists a finite set $\{c_i \in \mathcal{C} \mid i \in I\}$ such that $C' = \bigcup_{i \in I} \downarrow c_i$ and this equality can be derived. By Proposition 6 we then have $C = \bigcup_{i \in I} \exists\!\!\exists_x \downarrow c_i$. By Lemma 2 (3) we obtain $C = \bigcup_{i \in I} \downarrow \exists_x c_i$. This equality can be inferred by Rule (3).

($C = C_1 \cup C_2$) Obvious, using Rule (4).

Proposition 10 (Relative completeness for sat) The relation sat inductively defined by the system in Table 2 completely describes consistency in $\mathcal{L}$, i.e. for each element $C \in \mathcal{L}$, we derive $sat(C)$ iff $C \neq \mathit{False}$.

Proof By lemma 9 for some finite set of constraints $\{c_i \in \mathcal{C} \mid i \in I\}$, we can derive the equality $C = \bigcup_{i \in I} \downarrow c_i$. Then $C \neq \mathit{False}$ iff there exists $i \in I$ such that $c_i \neq \mathit{false}$. Hence $C \neq \mathit{False}$ iff $sat(C)$ can be derived, by using Rules (5) and (6).

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

Table 3: The transition system for Γ_clp.

5.3 The transition system

The operational semantics of Γ_clp is given by the transition relation $\longrightarrow$ defined by the rules in Table 3. The program $P \equiv D_1, D_2, \cdots, D_q$ is assumed to be fixed. The initial goal G in a transition step is supposed to have a consistent constraint, i.e. $con(G) \neq \mathit{False}$.

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 [25], we express this link by the context $\exists_\alpha(\delta_{y\alpha} \wedge \exists_x(\delta_{\alpha x} \wedge \ldots))$, where α is a variable which does not occur free in any program P or goal G (i.e. $\exists_\alpha c = c$ for all constraints c occurring in P or G), and α does not occur as a parameter (either as a formal parameter or as a actual parameter). Note that through the whole computation only one variable α is needed. This mechanism for treating procedure calls is much simpler and more elegant than the machinery of standardization apart used in (constraint) logic programming.

Parallel composition is modeled as interleaving.

Disjunction is modeled by the arbitrary choice of one of the alternatives which do not bring to inconsistency. There is no need to write explicitly the consistency check, because 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 $con(\exists_x G) \neq \mathit{False}$.

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 which do not contain procedure calls (are formed only by constraints), will be denoted by *TGoals*. The class of "final" goals, namely the set of G's such that $G \not\longrightarrow$, will be denoted by *FGoals*. Note that $TGoals \subseteq FGoals$ but in general the viceversa does not hold. A finite computation $G \longrightarrow^* G'$ is *successful* if $G' \in TGoals$, and it *fails* if $G' \in FGoals \setminus TGoals$. An infinite 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. A terminating computation (either successful of failed) is always called and-fair.

5.4 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 successful computations. We add the constraint *false* in order to avoid the possibility of an empty set, because the empty set is not an element of the denotational domain, and we aim at obtaining full correspondence with the denotational semantics. Note that if there is at least a successful derivation then *false* is already generated in the observables, because of the downward-closedness of the resulting constraint.

Definition 11 Given a program P, for every goal G we define

$$\mathcal{O}_P(G) \;=\; \bigcup\{con(G') \mid G \longrightarrow^* G' \text{ and } G' \in TGoals\} \;\cup\; \{false\}$$

Note that we obtain $\mathcal{O}_P(G) = \{false\}$ if and only if there are no successful computations. This corresponds to the concept of *universal failure*, according to the so-called notion of *don't know* nondeterminism: a failed computation branch is disregarded if there are successful computations. Note also that $\mathcal{O}_P(G) = \{false\}$ includes the case in which G has only infinite computations.

6 Denotational semantics

Our aim here is to give a compositional description of the constraints computed by a goal.

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 we will associate to each goal an element of the structure $\mathcal{P}_{\rm d}(\mathcal{C})$. We will call such an element a *process*.

In order to treat predicate definitions and recursion, we need to introduce the notion of (semantic) environment, namely a function mapping predicate names into elements of $\mathcal{P}_{\rm d}(\mathcal{C})$.

Definition 12 Let *Pred* be a set of predicate symbols. Define $Env = \{e \mid e : Pred \rightarrow \mathcal{P}_{\rm d}(\mathcal{C})\}$, with the ordering $e_1 \preceq e_2$ iff $\forall p.\ e_1(p) \subseteq e_2(p)$.

Since the ordering on *Env* is the pointwise extension of the ordering on a complete lattice, we have that $(Env, \preceq)$ is a complete lattice. The bottom $e_\perp$ is defined by $e_\perp(p) = False$ for all $p \in Pred$. For any countable set $E \subseteq Env$ we indicate the least upper bound of E by $\bigsqcup_\preceq E$. It is easy to verify that $\bigsqcup_\preceq E$ is defined by $(\bigsqcup_\preceq E)(p) = \bigcup_{e\in E} e(p)$ for all $p \in Pred$.

$$\mathcal{G}[\![c]\!]e = \downarrow c$$
$$\mathcal{G}[\![p(y)]\!]e = \exists_\alpha(\Delta_{y\,\alpha} \cap e(p)).$$
$$\mathcal{G}[\![G_1 \wedge G_2]\!]e = \mathcal{G}[\![G_1]\!]e \cap \mathcal{G}[\![G_2]\!]e$$
$$\mathcal{G}[\![G_1 \vee G_2]\!]e = \mathcal{G}[\![G_1]\!]e \cup \mathcal{G}[\![G_2]\!]e$$
$$\mathcal{G}[\![\exists_x G]\!]e = \exists_x \mathcal{G}[\![G]\!]e$$

Table 4: The interpretation function $\mathcal{G}$.

The equations defining the interpretation function $\mathcal{G}$ for goals, with respect to a given environment, are given in Table 4. Note that $\mathcal{G}$, applied to an environment, is a homomorphism from the algebra of Γ-clp goals into **C**.

6.1 The fixpoint operator

Finally, we need to define a function $\mathcal{D}_P$, associated to a program P, which transforms environments into environments, and plays the role of the one-step-inference operator of logic programming and contraint logic programming. We will show that such a function is continuous, and will use its least fixpoint to define the denotation of a goal wrt the given program P. Assuming, without loss of generality, that each predicate is defined exactly by one definition in P, we define such a function as follows:

Definition 13 On the complete lattice $(Env, \preceq)$, let $\mathcal{D}_P : Env \longrightarrow Env$ be the mapping defined by

$$\mathcal{D}_P(e)(p) = \exists_x(\Delta_{\alpha x} \cap \mathcal{G}[\![G_p]\!]e),$$

where $p(x)$:- G_p is the declaration for p in P.

We show now that the function $\mathcal{D}_P$ is continuous. We need the following lemmata, which express the continuity of $\mathcal{G}$ wrt the environments.

Lemma 14 For each goal G, $\mathcal{G}[\![G]\!]$ is monotonic on $(Env, \preceq)$. Namely, if $e_1 \preceq e_2$, then $\mathcal{G}[\![G]\!]e_1 \subseteq \mathcal{G}[\![G]\!]e_2$.

Proof By an easy induction on the structure of G.

Lemma 15 Let E be a directed set in $(Env, \preceq)$. Then, for all goals G, $\mathcal{G}[\![G]\!] \bigsqcup_\preceq E = \bigcup_{e \in E} \mathcal{G}[\![G]\!]e$.

Proof By induction on the structure of G.

$(G = c)$ $\quad \mathcal{G}[\![c]\!] \bigsqcup_\preceq E = \downarrow c = \bigcup_{e \in E} \mathcal{G}[\![c]\!]e$

$(G = p(y))$

$$
\begin{aligned}
& \mathcal{G}[\![p(y)]\!] \textstyle\bigsqcup_{\preceq} E \\
= \quad & \\
& \exists_\alpha (\Delta_{y\,\alpha} \cap (\textstyle\bigsqcup_{\preceq} E)(p)) \\
= \quad & \\
& \exists_\alpha (\Delta_{y\,\alpha} \cap \textstyle\bigcup_{e \in E} e(p)) \\
= \quad & \{\text{distributivity of } \cup \text{ wrt } \cap\} \\
& \exists_\alpha \textstyle\bigcup_{e \in E} (\Delta_{y\,\alpha} \cap e(p)) \\
= \quad & \{\text{Proposition 6}\} \\
& \textstyle\bigcup_{e \in E} \exists_\alpha (\Delta_{y\,\alpha} \cap e(p)) \\
= \quad & \\
& \textstyle\bigcup_{e \in E} \mathcal{G}[\![p(y)]\!] e.
\end{aligned}
$$

$(G = G_1 \wedge G_2)$

$$
\begin{aligned}
& \mathcal{G}[\![G_1 \wedge G_2]\!] \textstyle\bigsqcup_{\preceq} E \\
= \quad & \\
& (\mathcal{G}[\![G_1]\!] \textstyle\bigsqcup_{\preceq} E) \cap (\mathcal{G}[\![G_2]\!] \textstyle\bigsqcup_{\preceq} E) \\
= \quad & \{\text{induction hypothesis}\} \\
& (\textstyle\bigcup_{e \in E} \mathcal{G}[\![G_1]\!] e) \cap (\textstyle\bigcup_{e \in E} \mathcal{G}[\![G_2]\!] e) \\
= \quad & \{\text{since } \{\mathcal{G}[\![G_1]\!] e \mid e \in E\}, \{\mathcal{G}[\![G_1]\!] e \mid e \in E\} \text{ are directed set,} \\
& \quad \text{because } E \text{ is a directed set and } \mathcal{G}[\![G_1]\!], \mathcal{G}[\![G_1]\!] \text{ are monotonic} \\
& \quad \text{(Lemma 14)} \ \} \\
& \textstyle\bigcup_{e \in E} (\mathcal{G}[\![G_1]\!] e \cap \mathcal{G}[\![G_2]\!] e) \\
= \quad & \\
& \textstyle\bigcup_{e \in E} \mathcal{G}[\![G_1 \wedge G_2]\!] e.
\end{aligned}
$$

$(G = G_1 \vee G_2)$

$$
\begin{aligned}
& \mathcal{G}[\![G_1 \vee G_2]\!] \textstyle\bigsqcup_{\preceq} E \\
= \quad & \\
& (\mathcal{G}[\![G_1]\!] \textstyle\bigsqcup_{\preceq} E) \cup (\mathcal{G}[\![G_2]\!] \textstyle\bigsqcup_{\preceq} E) \\
= \quad & \{\text{induction hypothesis}\} \\
& (\textstyle\bigcup_{e \in E} \mathcal{G}[\![G_1]\!] e) \cup (\textstyle\bigcup_{e \in E} \mathcal{G}[\![G_2]\!] e) \\
= \quad & \\
& \textstyle\bigcup_{e \in E} (\mathcal{G}[\![G_1]\!] e \cup \mathcal{G}[\![G_2]\!] e) \\
= \quad & \\
& \textstyle\bigcup_{e \in E} \mathcal{G}[\![G_1 \vee G_2]\!] e.
\end{aligned}
$$

$(G = \exists_x G')$

$$
\begin{array}{ll}
& \mathcal{G}[\![\exists_x G']\!] \bigsqcup_{\preceq} E \\
= & \\
& \exists_x \mathcal{G}[\![G']\!] \bigsqcup_{\preceq} E \\
= & \{\text{induction hypothesis}\} \\
& \exists_x \bigcup_{e \in E} \mathcal{G}[\![G']\!] e \\
= & \{\text{Proposition 6}\} \\
& \bigcup_{e \in E} \exists_x \mathcal{G}[\![G']\!] e \\
= & \\
& \bigcup_{e \in E} \mathcal{G}[\![\exists_x G']\!] e.
\end{array}
$$

We are now able to prove the continuity of $\mathcal{D}_P$:

Proposition 16 For any program P, $\mathcal{D}_P$ is continuous, i.e. $\mathcal{D}_P(\bigsqcup_{\preceq} E) = \bigsqcup_{\preceq} \{\mathcal{D}_P(e) \mid e \in E\}$, for any directed set $E \subseteq Env$.

Proof Let $p(x)$:- G_p be the declaration of p in the program P. We have:

$$
\begin{array}{ll}
& \mathcal{D}_P(\bigsqcup_{\preceq} E)(p) \\
= & \{\text{Definition 13}\} \\
& \exists_x (\Delta_{\alpha x} \cap \mathcal{G}[\![G_p]\!] \bigsqcup_{\preceq} E) \\
= & \{\text{Lemma 15}\} \\
& \exists_x (\Delta_{\alpha x} \cap \bigcup_{e \in E} \mathcal{G}[\![G_p]\!] e) \\
= & \{\text{distributivity of } \cup \text{ wrt } \cap\} \\
& \exists_x \bigcup_{e \in E} (\Delta_{\alpha x} \cap \mathcal{G}[\![G_p]\!] e) \\
= & \{\text{Proposition 6}\} \\
& \bigcup_{e \in E} \exists_x (\Delta_{\alpha x} \cap \mathcal{G}[\![G_p]\!] e) \\
= & \\
& \bigcup_{e \in E} (\mathcal{D}_P(e))(p) \\
= & \\
& (\bigsqcup_{\preceq} \{\mathcal{D}_P(e) \mid e \in E\})(p).
\end{array}
$$

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 17 Define inductively the set $\{\bar{e}_i\}_{i \in \omega}$ as follows:

$$
\begin{array}{rcl}
\bar{e}_0 & = & e_\perp \\
\bar{e}_{i+1} & = & \mathcal{D}_P(\bar{e}_i)
\end{array}
$$

Proposition 18 For any program P, $\mathcal{D}_P$ has a least fixpoint $\mathbf{fix}\mathcal{D}_P = \bigsqcup_{\preceq} \{\bar{e}_i\}_{i \in \omega}$.

Proof Standard, by continuity of $\mathcal{D}_P$.

6.2 The model $\mathcal{G}_P$

We are now able to define the denotational model for the Γ_clp language.

Definition 19 Let P be a program. For a goal G, the denotation of G in P is defined by $\mathcal{G}_P[\![G]\!] = \mathcal{G}[\![G]\!]\mathbf{fix}\mathcal{D}_P$.

6.3 Correspondence between the operational and the denotational model

In this section we show that the observables $\mathcal{O}_P$ and the semantic model $\mathcal{G}_P$ coincide. The proof proceeds by establishing the two inclusions $\mathcal{G}_P[\![G]\!] \supseteq \mathcal{O}_P(G)$ (correctness) and $\mathcal{G}_P[\![G]\!] \subseteq \mathcal{O}_P(G)$ (completeness).

6.3.1 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 3 preserve the meaning of G, i.e. its semantics according to the equations in Table 4.

Lemma 20 Let P be a program and G a goal. Let $G \longrightarrow G'$ be a transition from G. Then $\mathcal{G}_P[\![G]\!] \supseteq \mathcal{G}_P[\![G']\!]$ holds.

Proof By induction on the structure of G.

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

$(G = p(y))$ The only transition step from $p(y)$ is $G \longrightarrow \exists_\alpha(\delta_{y\alpha} \wedge \exists_x(\delta_{\alpha x} \wedge G_p))$, where $p(x)$:- G_p is the declaration of p in P. Then:

$$
\begin{array}{ll}
 & \mathcal{G}_P[\![p(y)]\!] \\
= & \\
 & \exists\!\!\exists_\alpha(\Delta_{y\alpha} \cap \bigcup_{i\in\omega} \bar{e}_i(p)) \\
= & \\
 & \exists\!\!\exists_\alpha(\Delta_{y\alpha} \cap \exists\!\!\exists_x(\Delta_{\alpha x} \cap \bigcup_{i\in\omega, i\geq 1} \mathcal{G}[\![G_p]\!]\bar{e}_{i-1})) \\
= & \quad \{\text{Lemma 15}\} \\
 & \exists\!\!\exists_\alpha(\Delta_{y\alpha} \cap \exists\!\!\exists_x(\Delta_{\alpha x} \cap \mathcal{G}_P[\![G_p]\!])) \\
= & \\
 & \mathcal{G}_P[\![\exists_\alpha(\delta_{y\alpha} \wedge \exists_x(\delta_{\alpha x} \wedge G_p))]\!].
\end{array}
$$

$(G = G_1 \wedge G_2)$ Let $G_1 \longrightarrow G_1'$ (the symmetrical case is similar). We have:

$$
\begin{array}{ll}
 & \mathcal{G}_P[\![G_1 \wedge G_2]\!] \\
= & \\
 & \mathcal{G}_P[\![G_1]\!] \cap \mathcal{G}_P[\![G_2]\!] \\
\supseteq & \quad \{\text{induction hypothesis}\} \\
 & \mathcal{G}_P[\![G_1']\!] \cap \mathcal{G}_P[\![G_2]\!] \\
= & \\
 & \mathcal{G}_P[\![G_1' \wedge G_2]\!].
\end{array}
$$

$(G = G_1 \vee G_2)$ Let $G_1 \longrightarrow G_1'$ (the symmetrical case is similar). We have:

$$
\begin{array}{l}
\quad \mathcal{G}_P[\![G_1 \vee G_2]\!] \\
= \\
\quad \mathcal{G}_P[\![G_1]\!] \cup \mathcal{G}_P[\![G_2]\!] \\
\supseteq \\
\quad \mathcal{G}_P[\![G_1]\!] \\
\supseteq \quad \{\text{induction hypothesis}\} \\
\quad \mathcal{G}_P[\![G_1']\!].
\end{array}
$$

$(G = \exists_x G_1)$ Let G' be $\exists_x G_1'$ and $G_1 \longrightarrow G_1'$. Then:

$$
\begin{array}{l}
\quad d \in \mathcal{G}_P[\![\exists_x G_1']\!] \\
\Leftrightarrow \\
\quad \text{there exists } c \in \mathcal{G}_P[\![G_1']\!] \text{ such that } d \sqsubseteq \exists_x c \\
\Rightarrow \quad \{\text{induction hypothesis}\} \\
\quad \text{there exists } c \in \mathcal{G}_P[\![G_1]\!] \text{ such that } d \sqsubseteq \exists_x c \\
\Leftrightarrow \\
\quad d \in \mathcal{G}_P[\![\exists_x G_1]\!].
\end{array}
$$

Theorem 21 (Correctness) For each program P and goal G, $\mathcal{O}_P(G) \subseteq \mathcal{G}_P[\![G]\!]$ holds.

Proof Let $c \in \mathcal{O}_P(G)$. If $c = \mathit{false}$, then the assertion follows by the non-emptiness and downward-closedness of $\mathcal{G}_P[\![G]\!]$. On the other hand, if $c \neq \mathit{false}$, then $c \in \mathit{con}(G')$ for some $G' \in \mathit{TGoals}$ with $G \longrightarrow^* G'$. Since G' does not contain procedure calls, we have that $c \in \mathcal{G}_P[\![G']\!]$. By Lemma 20 we conclude $c \in \mathcal{G}_P[\![G']\!] \subseteq \mathcal{G}_P[\![G]\!]$.

6.3.2 Completeness

First we show the following lemmata.

Lemma 22 Let P be a program and G a goal. Let $G \longrightarrow G'$ be a transition from G. Then $\mathit{con}(G) \supseteq \mathit{con}(G')$.

Proof The proof proceeds by induction on the structure of G along the same lines as the proof of Lemma 20. The only different case is $G = p(y)$. For this case we observe that $\mathit{con}(p(y)) = \mathit{True} \supseteq \mathit{con}(G')$.

Lemma 23 For each goal G and for each $i \in \omega$, if $c \in \mathcal{G}[\![G]\!]\bar{e}_i$ and $c \neq \mathit{false}$, then there exists a derivation $G \longrightarrow^* G'$ such that $G' \in \mathit{TGoals}$ and $c \in \mathit{con}(G')$.

Proof We proceed by transfinite induction on i and on the structure s of G, with respect to the lexicographic ordering on $\langle i, s\rangle$ (i.e. $\langle i, s\rangle < \langle i', s'\rangle$ iff $i < i'$ or $(i = i'$ and $s < s'))$.

$(G = c)$ Obvious.

($G = p(y)$) Since $c \in \mathcal{G}[\![G]\!]\bar{e}_i$ and $c \neq \mathit{false}$, it must be the case that $i > 0$. Let $p(x)$:- G_p the declaration of p in P. Then, $\mathcal{G}[\![p(y)]\!]\bar{e}_i = \exists\!\!\exists_\alpha(\Delta_{y\alpha} \cap \bar{e}_i(p)) = \exists\!\!\exists_\alpha(\Delta_{y\alpha} \cap \exists\!\!\exists_x(\Delta_{\alpha x} \cap \mathcal{G}[\![G_p]\!]\bar{e}_{i-1})) = \mathcal{G}[\![\exists_\alpha(\delta_{y\alpha} \wedge \exists_x(\delta_{\alpha x} \wedge G_p))]\!]\bar{e}_{i-1}$. By the induction hypothesis on $i-1$ there exists a derivation $\exists_\alpha(\delta_{y\alpha} \wedge \exists_x(\delta_{\alpha x} \wedge G_p)) \longrightarrow^* G'$ with $G' \in \mathit{TGoals}$ and $c \in \mathit{con}(G')$. Then, $p(y) \longrightarrow \exists_\alpha(\delta_{y\alpha} \wedge \exists_x(\delta_{\alpha x} \wedge G_p)) \longrightarrow^* G'$ is the desired derivation from $p(x)$.

($G = G_1 \wedge G_2$) If $c \in \mathcal{G}[\![G_1 \wedge G_2]\!]\bar{e}_i$, then $c \in \mathcal{G}[\![G_1]\!]\bar{e}_i$ and $c \in \mathcal{G}[\![G_2]\!]\bar{e}_i$. By the induction hypothesis, there exist derivations $G_j \longrightarrow^* G'_j$ with $G'_j \in \mathit{TGoals}$, for both $j = 1$ and $j = 2$. Thus, $c \in \mathit{con}(G'_1) \cap \mathit{con}(G'_2) = \mathit{con}(G'_1 \wedge G'_2)$. Since $c \neq \mathit{false}$ holds, and during a derivation the constraint associated with the goals can only increase (Lemma 22), the two derivations can be combined into a derivation $G_1 \wedge G_2 \longrightarrow^* G'_1 \wedge G'_2$. In fact for each intermediate goal G''_1 in the derivation of G_1, and each intermediate goal G''_2 in the derivation of G_2, $\mathit{con}(G'_1) \subseteq \mathit{con}(G''_1)$ and $\mathit{con}(G'_2) \subseteq \mathit{con}(G''_2)$ hold, hence $c \in \mathit{con}(G''_1) \cap \mathit{con}(G''_2)$, therefore $\mathit{sat}(\mathit{con}(G''_1 \wedge G''_2))$ can be derived by the system in Table 2.

($G = G_1 \vee G_2$) The case $G_1 \vee G_2 \in \mathit{TGoals}$ is obvious. For the case $G_1 \vee G_2 \notin \mathit{TGoals}$, observe that $c \in \mathcal{G}[\![G_1 \vee G_2]\!]\bar{e}_i$ implies either $c \in \mathcal{G}[\![G_1]\!]\bar{e}_i$ or $c \in \mathcal{G}[\![G_2]\!]\bar{e}_i$. By the induction hypothesis, there exists a derivation either from G_1 or from G_2 ending in a terminal goal G' such that $c \in \mathit{con}(G')$. Therefore we derive $G_1 \vee G_2 \longrightarrow^* G'$.

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

Theorem 24 (Completeness) For each program P and goal G, $\mathcal{G}_P[\![G]\!] \subseteq \mathcal{O}_P(G)$ holds.

Proof Let $c \in \mathcal{G}_P[\![G]\!]$. The case $c = \mathit{false}$ is obvious. If $c \neq \mathit{false}$, then we have:

$$
\begin{array}{ll}
 & c \in \mathcal{G}_P[\![G]\!] \\
\Leftrightarrow & \{\text{ Lemma 15}\} \\
 & c \in \bigcup_{i\in\omega} \mathcal{G}[\![G]\!]\bar{e}_i \\
\Leftrightarrow & \\
 & \text{there exists } i \text{ such that } c \in \mathcal{G}[\![G]\!]\bar{e}_i \\
\Rightarrow & \{\text{ Lemma 23 }\} \\
 & G \longrightarrow^* G' \text{ for some } G' \in \mathit{TGoals} \text{ and } c \in \mathit{con}(G') \\
\Leftrightarrow & \\
 & c \in \mathcal{O}_P(G).
\end{array}
$$

7 Declarative semantics

In this section we define a declarative (or model-theoretic) characterization of the observables of a goal. The declarative interpretation of a program is defined with respect to an environment:

Definition 25 A clause $p(x)$:- G is true in an environment e, notation $e \models p(x)$:- G, if $e(p) = \exists_x e(p)$ and $\mathcal{G}[\![G]\!]e = \mathcal{G}[\![p(x)]\!]e$. A program P is true in an environment e, notation $e \models P$, if $e \models p(x)$:- G, for all declarations $p(x)$:- G of P.

Note that the above definition uses the fact that we have only one declaration of $p(x)$ in P, so we can interpret :- safely as logical equivalence. Furthermore, since the information about the variable x in the meaning of a goal $p(x)$ with respect to an environment e is derived from the information about the variable α in $e(p)$ ($\mathcal{G}[\![p(x)]\!]e = \exists_\alpha(\Delta_{\alpha x} \cap e(p))$), we have to require that a valid environment e for a clause for $p(x)$ does not contain information about x, that is, $e(p) = \exists_x e(p)$. (Note that $\exists_\alpha(\Delta_{\alpha x} \cap e(p))$ corresponds intuitively to renaming the variable α occurring in $e(p)$ into x.) We have that an environment e is a valid interpretation of a program P if and only if e is a fixpoint of the operator $\mathcal{D}_P$:

Theorem 26 For every program P, environment e, we have $e \models P$ if and only if $\mathcal{D}_P(e) = e$.

Proof We need the following lemma:

Lemma 27 For any $C \in \mathbf{C}$ such that $\exists_x C = C$, we have that $\exists_x(\Delta_{\alpha x} \cap \exists_\alpha(\Delta_{x\alpha} \cap C)) = C$.

Proof We use the fact that $\mathbf{C}$ is a cylindric constraint system ordered by set-inclusion.

($\subseteq$) We have $\Delta_{\alpha x} \cap \exists_\alpha(\Delta_{x\alpha} \cap C)) \subseteq C$ by Axiom (vii). So by Axiom (ii) we derive that $\exists_x(\Delta_{\alpha x} \cap \exists_\alpha(\Delta_{x\alpha} \cap C)) \subseteq \exists_x C = C$.

($\supseteq$) By the hypothesis about C and Axiom (i) we have that $\Delta_{x\alpha} \cap \exists_x C \subseteq \exists_\alpha(\Delta_{x\alpha} \cap \exists_x C) = \exists_\alpha(\Delta_{x\alpha} \cap C)$. So by Axiom (ii) we derive that $\exists_x(\Delta_{\alpha x} \cap \Delta_{x\alpha} \cap \exists_x C) \subseteq \exists_x(\Delta_{\alpha x} \cap \exists_\alpha(\Delta_{x\alpha} \cap C))$. Next we observe that

$$
\begin{array}{ll}
& \exists_x(\Delta_{\alpha x} \cap \Delta_{x\alpha} \cap \exists_x C) \\
= & \\
& \exists_x(\Delta_{\alpha x} \cap \exists_x C) \\
= & \{\text{by Axiom (iii)}\} \\
& \exists_x \Delta_{\alpha x} \cap \exists_x C \\
= & \{\exists_x \Delta_{\alpha x} = \mathit{True}\} \\
& \exists_x C \\
= & \\
& C.
\end{array}
$$

Note that $\exists_x \Delta_{\alpha x} = \exists_x(\Delta_{\alpha x} \cap \Delta_{x\alpha}) = \delta_{\alpha\alpha} = \mathit{True}$ by Axiom (vi).

We can now prove Theorem 26. Let $e \models p(x)$:- G, with $p(x)$:- G a clause of P. Then $\mathcal{G}[\![p(x)]\!]e = \mathcal{G}[\![G]\!]e$ and $\exists_x e(p) = e(p)$. Hence

$$\begin{aligned}
& \mathcal{D}_P(e)(p) \\
= \quad & \{\text{by definition}\} \\
& \exists_x(\Delta_{x\alpha} \cap \mathcal{G}[\![G]\!]e) \\
= \quad & \{\mathcal{G}[\![p(x)]\!]e = \mathcal{G}[\![G]\!]e\} \\
& \exists_x(\Delta_{x\alpha} \cap \mathcal{G}[\![p(x)]\!]e) \\
= \quad & \{\text{by definition}\} \\
& \exists_x(\Delta_{x\alpha} \cap \exists_\alpha(\Delta_{x\alpha} \cap e(p))) \\
= \quad & \{\text{by } \exists_x e(p) = e(p) \text{ and Lemma 27}\} \\
& e(p)
\end{aligned}$$

Next let $\mathcal{D}_P(e) = e$. Let $p(x)$:- G be a clause of P. Then:

$$\begin{aligned}
& \mathcal{G}[\![p(x)]\!]e \\
= \quad & \{\text{by definition}\} \\
& \exists_\alpha(\Delta_{\alpha x} \cap e(p)) \\
= \quad & \{\mathcal{D}_P(e) = e\} \\
& \exists_\alpha(\Delta_{\alpha x} \cap \exists_x(\Delta_{x\alpha} \cap \mathcal{G}[\![G]\!]e)) \\
= \quad & \{\text{by Lemma 27 and } \exists_\alpha \mathcal{G}[\![G]\!]e = \mathcal{G}[\![G]\!]e\} \\
& \mathcal{G}[\![G]\!]e.
\end{aligned}$$

(A straightforward induction on the complexity of the goal G shows that $\exists_\alpha \mathcal{G}[\![G]\!]e = \mathcal{G}[\![G]\!]e$, under the assumption that $\exists_\alpha c = c$ for every constraint occurring in G and α does not occur in procedure calls.) Moreover, since $e(p) = \mathcal{D}_P(e)(p) = \exists_x(\Delta_{x\alpha} \cap \mathcal{G}[\![G]\!]e)$, it follows immediately that $\exists_x e(p) = e(p)$. Summarizing the above we thus conclude that $e \models p(x)$:- G.

We consider as possible semantical consequences of a program implications of the form $c \Rightarrow G$. The interpretation of $\Rightarrow$ corresponds to the notion of entailment in **C**.

Definition 28 The implication $c \Rightarrow G$ is a semantical consequence of a program P, notation $P \models c \Rightarrow G$, if for all environments e such that $e \models P$ we have that $\mathcal{G}[\![c]\!]e \subseteq \mathcal{G}[\![G]\!]e$.

We thus arrive at the following declarative characterization of the observables of a goal:

Theorem 29 For every program P, constraint c, goal G we have that $P \models c \Rightarrow G$ if and only if $c \in \mathcal{O}_P(G)$.

Proof We have:

$$c \in \mathcal{O}_P(G)$$

$$\Leftrightarrow \quad \{\mathcal{O}_P(G) \text{ is downward-closed and Theorems 21 and 24}\}$$

$$\mathcal{G}[\![c]\!]e \subseteq \mathcal{G}[\![G]\!]\mathrm{fix}\mathcal{D}_P$$

$$\Leftrightarrow \quad \{(\Rightarrow) \text{ monotonicity of } \mathcal{G}[\![G]\!]\}$$

$$\mathcal{G}[\![c]\!]e \subseteq \mathcal{G}[\![G]\!]e, \text{ for every } e \text{ such that } \mathcal{D}_P(e) = e$$

$$\Leftrightarrow \quad \{\text{Theorem 26}\}$$

$$\mathcal{G}[\![c]\!]e \subseteq \mathcal{G}[\![G]\!]e, \text{ for every } e \text{ such that } e \models P$$

$$\Leftrightarrow$$

$$P \models c \Rightarrow G.$$

8 A model for Negation

We consider now the possibility of introducing a construct for negation. The treatment of negation we propose is different from the other constructive and non-constructive approaches in (constraint) logic programming. The distinguishing feature is its "algebraicity", which derives from the use of constraint systems in the style of [24], as opposite to the "interpretative" style of [18]. The use of a lattice structure representing directly the constraints and the entailment relation, rather than some external interpretation structure, allowed us in previous sections to define a distributive lattice as the domain of goals' denotation. Now we are going to use some well-known results of the theory of lattices to model negation.

8.1 Heyting negation

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 define a notion of negation on the structure $\mathbf{C}$. The first idea would be to define $\neg C$ as $\mathcal{C} \setminus C$, but this operation is not well-defined on $\mathbf{C}$ since the resulting set is in general not downward closed. Furthermore this interpretation of negation does not allow a declarative characterization. Consider the following example:

Example 30 Let G be the goal $x = a$, and P be the empty program. Then, for example, $\mathit{true} \in \mathcal{C} \setminus \mathcal{O}_P(x = a)$, however, obviously $P \models \mathit{true} \Rightarrow \neg x = a$ does not hold.

Another possibility is to use a form of negation which is already implicit in our structure $\mathbf{C}$: the Heyting negation. We will see that this form of negation allows for a simple operational definition in terms of our transition system, and that it actually corresponds to an extension of the well-known concept of *negation as failure*. Also the negation rule introduced in [10], namely the *negation as instantiation*, can be formulated in terms of Heyting negation in a very simple way.

Heyting algebras have been extensively studied in mathematics for modeling intuitionistic logic. Recently, these structures have been investigated in connection with the theory of topological spaces ([28]) and with the theory of programming languages ([1]).

Definition 31 ([16]) A *complete Heyting algebra* (cHa) is a complete lattice $\mathcal{L}$ which satisfies the following infinite distributive law:

$$glb(x, lub(Y)) = lub(\{glb(x, y) \mid y \in Y\}) \tag{1}$$

for all element $x \in \mathcal{L}$ and all sets $Y \subseteq \mathcal{L}$.

The property (1) is called *frame distributivity* to distinguish it from the finite distributivity. Since the set intersection is distributive also wrt the infinite set union, we have that:

Remark 32 $\mathbf{C}$ is a complete Heyting algebra.

The fact that $\mathbf{C}$ is a Heyting algebra can be also characterized by the following way, which makes the intuitionistic aspect more evident.

Proposition 33 (See [28]) For every $C_1, C_2 \in \mathcal{P}_{\mathrm{d}}(\mathcal{C})$ there is an element $C_1 \rightarrow C_2$ in $\mathcal{P}_{\mathrm{d}}(\mathcal{C})$ satisfying

$$\forall C \in \mathcal{P}_{\mathrm{d}}(\mathcal{C}).\ \ C \subseteq C_1 \rightarrow C_2 \text{ iff } C \cap C_1 \subseteq C_2.$$

The operation $\rightarrow$ is called Heyting implication. The standard way to define it, in a lattice whose elements are sets and the ordering relation is set-inclusion, is:

$$C_1 \rightarrow C_2 = \bigcup\{C \in \mathcal{P}_{\mathrm{d}}(\mathcal{C}) \mid C \cap C_1 \subseteq C_2\}.$$

Using the Heyting implication, we define the *Heyting negation* in the standard way:

$$\neg C = C \rightarrow \mathit{False}$$

Note that for the Heyting negation, $\neg \mathit{True} = \mathit{False}$ and $\neg \mathit{False} = \mathit{True}$ hold. Moreover, $C \cap \neg C = \mathit{False}$ and the following De-Morgan rule

$$\neg C_1 \cap \neg C_2 = \neg(C_1 \cup C_2) \tag{2}$$

always hold, whereas $C \cup \neg C = \mathit{True}$ and the other De-Morgan rule $\neg C_1 \cup \neg C_2 = \neg(C_1 \cap C_2)$ do not hold in general.

We now introduce in the language a construct for negation and we model it by means of the Heyting negation. Due to the non-monotonicity of the negation operator, in general the denotational semantics of previous section cannot be defined when negative goals occur in the bodies of the clauses, because the function $\mathcal{D}$ is not guaranteed to be continuous. Thus, in this paper, we consider *positive* programs only. Less restrictive solutions to this problem are however applicable, like for instance *stratification* ([2]).

The function *con* of Definition 7 is extended to negative goals in the following way:

$$con(\neg G) = \neg con(G).$$

8.2 Operational semantics, observables and denotation 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:

$$\textbf{Negation} \quad \frac{G \longrightarrow G'}{\neg G \longrightarrow \neg G'} \tag{3}$$

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 34 Consider the program

$$\begin{array}{lll} p(x) & :- & q(x) \vee r(x), \\ q(x) & :- & x = 0, \\ r(x) & :- & x = 1, \end{array}$$

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

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

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

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

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

However, the negation rule by itself is not unsound. It only requires the introduction of an appropriate 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 35 If G is a positive goal, namely a goal generated by the grammar in Table 1, then define

$$\mathcal{O}_P(\neg G) = \bigcap\{con(\neg G') \mid \neg G \longrightarrow^* \neg G' \text{ and } G' \in TGoals\}$$

Note that if there are no $G' \in TGoals$ such that $G \longrightarrow^* G'$, then $\mathcal{O}_P(\neg G) = True$ because the intersection of empty class of sets is the entire universe $\mathcal{C}$.

Example 36 Consider the program and the constraint system of Example 34. Then we have $\neg con(x = 0) \cap \neg con(x = 1) = \downarrow\{x = 1\} \cap \downarrow\{x = 0\}$, therefore $\mathcal{O}_P(\neg p(x)) = False$.

Example 37 Consider the program of Example 34, with the constraint system of Figure 1. Then we have $\neg con(x = 0) \cap \neg con(x = 1) = (\downarrow\{x = 1\} \cup \downarrow\{x = 2\}) \cap (\downarrow\{x = 0\} \cup \downarrow\{x = 2\}) = \downarrow\{x = 2\}$, therefore $\mathcal{O}_P(\neg p(x)) = \downarrow\{x = 2\}$.

If we want to use $\neg$ as a free operator, namely to extend the grammar of goals as follows:

$$G ::= c \mid p(x) \mid G \wedge G \mid G \vee G \mid \exists_x G \mid \neg G \tag{4}$$

then it is convenient to adopt a structured definition of observables, as given in the following:

Definition 38 If G is a goal generated by the grammar in (4), define $\mathcal{O}_P(G)$ by structural induction on G as follows:

$$\begin{array}{lcl}
\mathcal{O}_P(c) & = & \mathcal{O}_P(c) \\
\mathcal{O}_P(p(x)) & = & \mathcal{O}_P(p(x)) \\
\mathcal{O}_P(G_1 \wedge G_2) & = & \mathcal{O}_P(G_1) \cap \mathcal{O}_P(G_2) \\
\mathcal{O}_P(G_1 \vee G_2) & = & \mathcal{O}_P(G_1) \cup \mathcal{O}_P(G_2) \\
\mathcal{O}_P(\exists_x G) & = & \exists_x \mathcal{O}_P(G) \\
\mathcal{O}_P(\neg G) & = & \neg \mathcal{O}_P(G).
\end{array}$$

Definitions 35 and 38 are compatible, as shown by the following proposition:

Proposition 39 The extension of $\mathcal{O}_P$ in Definition 38 is a consistent extension of the mapping $\mathcal{O}_P$ in Definition 35.

Proof It is sufficient to show that Definition 35 and Definition 38 coincide for $\neg G$ when G is a positive goal. The cases $G = c$, $G = p(y)$, $G = G_1 \wedge G_2$ and $G = \exists_x G'$ are obvious. For $G = G_1 \vee G_2$ we have, according to Definition 35:

$$\begin{array}{rcl}
\mathcal{O}_P(\neg G) & = & \bigcap\{con(\neg G') \mid G_1 \longrightarrow^* G' \text{ for some } G' \in TGoals\} \\
& \cap & \bigcap\{con(\neg G') \mid G_2 \longrightarrow^* G' \text{ for some } G' \in TGoals\} \\
& \cap & \{con(\neg G) \mid G \in TGoals\}.
\end{array}$$

On the other hand, according to Definition 38 we have:

$$\begin{array}{rcll}
\mathcal{O}_P(\neg G) & = & \neg\ (& \bigcup\{con(G') \mid G_1 \longrightarrow^* G' \text{ for some } G' \in TGoals\} \\
& & \cup & \bigcup\{con(G') \mid G_2 \longrightarrow^* G' \text{ for some } G' \in TGoals\} \\
& & \cup & \{con(G) \mid G \in TGoals\} \\
& & \cup & \{false\} \qquad).
\end{array}$$

The identity of these two definitions follows then by the De-Morgan rule in (2).

Concerning the denotational semantics, the natural way to interpret negative goals is to assign to $\neg G$ the Heyting negation of the process associated to the corresponding positive goal G; namely, we extend the function $\mathcal{G}_P$ as follows:

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

For positive programs, this simple extension of the denotational semantics $\mathcal{G}_P$ is a good semantics for negative goals. In fact, it is immediate to verify that the correspondence with the observables is maintained:

Proposition 40 For every positive program P and for every goal G generated by the grammar in (4), we have: $\mathcal{G}_P[\![\neg G]\!] = \mathcal{O}_P(\neg G)$.

Proof Immediate from the extended definition of $\mathcal{O}_P(\neg G)$ (Definition 38). For the cases $G = c$ and $G = p(y)$ Theorems 21 and 24 apply. For the other cases, just observe that $\mathcal{O}_P$ maps the syntactical operators in the same semantical operators as $\mathcal{G}_P$.

8.3 Relationship with NAF and NAI

The treatment of negative information by means of a transition rule and a denotational semantics based on the notion of Heyting negation, which we have described above, allows for an extension of the notion of negation associated to some inference rules such as *negation as failure* (NAF) or *negation as instantiation* (NAI). This is possible thanks to the ability to define step-by-step the evolution of a negative goal just as in the case of positive goals. However, we can construct the answers for the negative goal only at a declarative level by suitably combining the answers obtained operationally (i.e. by the transition rule). It is well known that *constructive negation* ([6, 19]) does construct answers for negative goals directly at the operational level. This is the main difference between the latter and our negation. On the contrary, the negation rules above mentioned are not able to construct answers at all; they are only tests for validity of some formulas (namely universally quantified formulas for NAF and existentially quantified formulas for NAI). Therefore, our negation represents an extension of these rules, even if we cannot properly call it a constructive extension.

8.3.1 Negation as failure

Recall that negation as failure is formulated as follows:

$$\textbf{NAF rule} \quad \frac{\text{all and-fair computations of } G \text{ fail}}{\neg G}$$

for a positive goal G. In order to describe the premise of this rule in terms of the observables derivable from our transition system, we have to include information about termination. A natural formulation of the premise of the rule, in fact, would be

$$\mathcal{O}_P(\neg G) = \mathit{True},$$

but this does not correspond, in general, to say that all and-fair derivations from G fail. Consider the following example:

Example 41 Let P be the program consisting only of the clause $p(x)$:- $p(x)$. Then we have only one derivation, which is and-fair, and not failing, from $p(x)$:

$$p(x) \longrightarrow p(x) \longrightarrow p(x) \longrightarrow \ldots$$

(we abstract here from the parameter-passing mechanism). However, since there are no derivations ending in a terminal goal, we have $\mathcal{O}_P(\neg G) = \mathit{True}$.

In order to achieve full correspondence with negation as failure, we have to take into account also infinite computations. Following [11], we define for a positive goal G

$$\begin{array}{rcll} \mathcal{O}_P^{\infty}(G) & = & \bigcup\{\mathit{con}(G') \mid & G \longrightarrow^* G' \text{ and } G' \in \mathit{TGoals}\} \\ & \cup & \bigcup\{\bigcap_i \mathit{con}(G_i) \mid & G \longrightarrow G_1 \longrightarrow G_2 \longrightarrow \ldots \longrightarrow G_i \longrightarrow \ldots \\ & & & \text{is an infinite and-fair computation}\ \} \\ & \cup & \{\mathit{false}\}. & \end{array}$$

Namely, we add to the observables the limit results of the infinite computations. Note that $\bigcap_i \mathit{con}(G_i)$ is an element of $\mathcal{P}_d(\mathcal{C})$, since the sets $\mathit{con}(G_i)$'s are non-empty and downward-closed. In [11] it is also shown that this notion of observables can be modeled denotationally by considering the greatest fixpoint of the operator $\mathcal{D}_P$.

It is immediate now to see that the premise of the negation-as-failure rule corresponds to $\mathcal{O}_P^\infty(G) = \mathit{False}$, under the assumption that *False* cannot be obtained as the limit of a decreasing chain of constraints different from *False*. This assumption is customary for the cylindric-algebraic approach to constraint systems. More specifically, it is usually required that *false* is a *finitary* element, i.e. if we have a decreasing chain $\{c_i \in \mathcal{C} \mid i \in \omega\}$, then $\sqcap_{i\in\omega} c_i = \mathit{false}$ implies that there exists $i \in \omega$ such that $c_i = \mathit{false}$. This corresponds to require the *compactness* of the entailment relation: if a contradiction can be derived from an infinite set of hypothesis, then it can be derived from a finite subsets of it. In the rest of this section, we assume this condition to hold on the underlying system $\mathcal{C}$. In our setting we can therefore reformulate the NAF rule as follows:

$$\frac{\mathcal{O}_P^\infty(G) = \mathit{False}}{\neg G \longrightarrow \mathit{true}}$$

where $\longrightarrow$ is the transition relation. We would like, however, to have a formulation of the premise in terms of the observables of the negated goal derived directly from the transition system. Namely we want to define directly $\mathcal{O}_P^\infty(\neg G)$ in terms of transitions from $\neg G$, and in such a way that $\mathcal{O}_P^\infty(G) = \mathit{False}$ iff $\mathcal{O}_P^\infty(\neg G) = \mathit{True}$. The natural way to define $\mathcal{O}_P^\infty(\neg G)$ would be to add information about infinite computations to the observables of a negated goal, namely to extend Definition 35 as follows:

$$\begin{array}{rcll}
\mathcal{O}_P^\infty(\neg G) & = & & \bigcap\{\mathit{con}(\neg G') \mid \ \neg G \longrightarrow^* \neg G' \text{ and } G' \in \mathit{TGoals}\} \\
& & \cap & \bigcap\{\bigcup_i \mathit{con}(\neg G_i) \mid \ \neg G \longrightarrow \neg G_1 \longrightarrow \neg G_2 \longrightarrow \ldots \neg G_i \longrightarrow \ldots \\
& & & \qquad \text{is an infinite and-fair computation }\}.
\end{array}$$

We show now that with this definition we obtain

$$\mathcal{O}_P^\infty(\neg G) = \neg\mathcal{O}_P^\infty(G).$$

It is not immediate to derive the above equality, because it requires to show that $\neg\bigcap_i \mathit{con}(G_i) = \bigcup_i \mathit{con}(\neg G_i)$, and to this purpose we cannot use the infinite De-Morgan law $\bigcup_i \neg C_i = \neg\bigcap_i C_i$, because this does not hold in general. In our case, however, we are dealing with special classes of C_i's, and we will show that for these classes the property holds, under the assumption that, for every decreasing chain $\{c_i \in \mathcal{C} \mid i \in \omega\}$, there exists the greatest lower bound $\sqcap_{i\in\omega} c_i$ and it is an element of $\mathcal{C}$. Such a requirement is customary for constraint systems in the approach of [25], and we assume it in the rest of this section.

Proposition 42 Let $\{C_i \in \mathcal{P}(\mathcal{C}) \mid i \in \omega\}$ be a decreasing chain of finitary sets, i.e. $C_0 \supseteq C_1 \supseteq \ldots C_i \supseteq \ldots$, and C_i finitary for every $i \in \omega$. Then $\bigcup_{i\in\omega} \neg C_i = \neg\bigcap_{i\in\omega} C_i$ holds.

Proof Let us first observe that, for each set $C \in \mathcal{P}(\mathcal{C})$, we have $\neg C = \{d \in \mathcal{C} \mid \forall c \in C.\ c \sqcap d = \mathit{false}\}$.

($\subseteq$) Assume $d \in \bigcup_{i\in\omega} \neg C_i$. Then, there exists $i \in \omega$ such that, for all $c \in C_i$, $c \sqcap d = \mathit{false}$ holds. Therefore, for all $c \in \bigcap_{i\in\omega} C_i$, $c \sqcap d = \mathit{false}$ holds as well.

($\supseteq$) Assume, by contradiction, that there exists $d \in \mathcal{C}$ such that $d \in \neg\bigcap_{i\in\omega} C_i$ and $d \notin \bigcup_{i\in\omega} \neg C_i$. By the latter we infer that for every $i \in \omega$ there exists an element $c_i \in C_i$ such that $c_i \sqcap d \neq \mathit{false}$. We construct now a decreasing chain $\{d_j \in \mathcal{C} \mid j \in \omega\}$ such that $d_j \sqcap d \neq \mathit{false}$ for every $j \in \omega$.

Let d_0 be a maximal element of C_0 such that for an infinite set of indexes $I_0 \subseteq \omega$, $c_i \sqsubseteq d_0$ holds for all $i \in I_0$. Such an element d_0 exists because C_0 is finitary and $c_i \in C_0$ for each $i \in \omega$. Observe that $d_0 \sqcap d \neq \mathit{false}$ holds, because $c_i \sqcap d \neq \mathit{false}$ and $c_i \sqsubseteq d_0$ (for every $i \in \omega$).

Repeat now the construction so to generate, for every $j \geq 1$, an infinite set of indexes I_j, an index $i_j \in \omega$, with $i_j > i_{j-1}$, and an element $d_j \in C_{i_j}$, as follows: Let i_j be the least index in $I_{j-1} \setminus \{i_{j-1}\}$. Let d'_j be a maximal element of C_{i_j} such that, for an infinite set of indexes $I_j \subseteq I_{j-1}$, $c_i \sqsubseteq d'_j$ holds for all $i \in I_j$. (Again, such an element exists because C_{i_j} is finitary and $c_i \in C_{i_j}$ holds for each $i \in I_{j-1}$.) Observe that $d'_j \sqcap d \neq \mathit{false}$ holds. Define now $d_j = d'_j \sqcap d_{j-1}$, and observe that $c_i \sqsubseteq d_j$ holds (for all $i \in I_j$). Therefore, $d_j \sqcap d \neq \mathit{false}$. Furthermore, we have $d_j \sqsubseteq d_{j-1}$.

Consider now the element $\sqcap_{j\in\omega} d_j$. By construction, $\sqcap_{j\in\omega} d_j \in \bigcap_{j\in\omega} C_{i_j}$ because each C_{i_j} is downward closed. Furthermore, $\bigcap_{j\in\omega} C_{i_j} = \bigcap_{i\in\omega} C_i$ because $\{C_i \in \mathcal{P}(\mathcal{C}) \mid i \in \omega\}$ is a decreasing chain. Therefore, by the assumption $d \in \neg \bigcap_{i\in\omega} C_i$, we derive $d \sqcap (\sqcap_{j\in\omega} d_j) = \sqcap_{j\in\omega}(d \sqcap d_j) = \mathit{false}$. But, since false is a finitary constraint, this implies $d \sqcap d_j = \mathit{false}$ for some $j \in \omega$, which gives the contradiction.

Corollary 43 Let $G_0 \longrightarrow G_1 \longrightarrow G_2 \longrightarrow \ldots \longrightarrow G_i \longrightarrow \ldots$ be an infinite computation. Then $\neg \bigcap_{i\in\omega} \mathit{con}(G_i) = \bigcup_{i\in\omega} \mathit{con}(\neg G_i)$.

Proof From Lemma 9 we know that every set $\mathit{con}(G_i)$ is finitary, and from Lemma 22 we know that the sets $\mathit{con}(G_i)$'s form a decreasing chain. Then apply Proposition 42.

Corollary 44 For every positive program P and goal G we have $\mathcal{O}_P^\infty(\neg G) = \neg\mathcal{O}_P^\infty(G)$.

Proof The proof is a simple extension of the proof of Proposition 39, using additionally the property expressed by previous corollary.

In our setting we can therefore reformulate the NAF rule as follows:

$$\frac{\mathcal{O}_P^\infty(\neg G) = \mathit{True}}{\neg G \longrightarrow \mathit{true}}.$$

Our definition of negation has a correspondence also with an extension of negation as failure, which has been defined by Shepherdson in [23]. This is formulated as follows:

$$\textbf{Shepherdson's rule} \quad \frac{\text{all and-fair computations of } G\theta \text{ fail}}{\neg G \text{ succeeds with answer substitution } \theta}$$

for a positive goal G. Taking into account that $A\theta$ can be expressed in clp as $c_\theta \wedge A$, where c_θ is an equality constraint corresponding to the substitution θ, we can reformulate Shepherdson's inference rule as follows:

$$\frac{\mathcal{O}_P^\infty(c \wedge G) = \mathit{False}}{\neg G \longrightarrow c}$$

or, equivalently

$$\frac{\mathcal{O}_P^\infty(\neg(c \wedge G)) = \mathit{True}}{\neg G \longrightarrow c}$$

Example 45 Consider the program

$$p(x) \quad :\!- \quad \exists_y (x = s(y) \wedge p(y))$$

in a constraint system whose elements are: $true, x = 0, x \neq 0, \exists_y x = s(y), x = s(0), x \neq s(0), \exists_y x = s(s(y)), x = s(s(0)), \ldots, x = s^\omega, false$, with the obvious ordering.

Observe that the element $x = s^\omega$, namely $\sqcap_i \exists_y x = s^i(y)$, must exist and be distinguished from $false$, because of the requirement that $false$ is finitary. Observe also that we cannot have a constraint of the form $x \neq s^\omega$, because $x \neq s^\omega \sqcap x = s^\omega$ should give $false$, and this would imply that for some $i \in \omega$, $x \neq s^\omega \sqcap x = s^i(y)$ gives $false$, which is unnatural.

Using the rule for negation in (3) we derive (abstracting from the parameter-passing mechanism):

$$\begin{array}{rcl} p(x) & \longrightarrow & \exists_y (x = s(y) \wedge p(y)) \\ & \longrightarrow & \exists_y (x = s^2(y) \wedge p(y)) \\ & \longrightarrow & \ldots \\ & \longrightarrow & \exists_y (x = s^i(y) \wedge p(y)) \\ & \longrightarrow & \ldots \end{array}$$

which is the only computation from $p(x)$. Hence we have $\mathcal{O}_P^\infty(\neg p(x)) = \bigcup_{i \in \omega} \neg \downarrow (\exists_y x = s^i(y)) = \neg \bigcap_{i \in \omega} \downarrow (\exists_y x = s^i(y)) = \neg \downarrow (x = s^\omega) = \{x = 0, x = s(0), x = s(s(0)), \ldots, false\} \neq True$. Note that the premise of the NAF rule is not satisfied either for the goal $:\!- p(x)$, since the (only) computation from $:\!- p(x)$ is infinite and and-fair.

On the other hand, the premise of Shepherdson's rule is satisfied for the goal $:\!- p(x)\{x/s^i(0)\}$, for any $i \in \omega$. Correspondingly we have $\mathcal{O}_P^\infty(\neg(x = s^i(0) \wedge p(x))) = True$, because the only computation from $\neg(x = s^i(0) \wedge p(x))$ ends, after i steps, in a non-terminal goal.

8.3.2 Negation as instantiation

Negation as instantiation (NAI) is based on the following idea: Consider a Herbrand universe enriched with infinitely many constant symbols. Assume that all and-fair computations from a positive goal G either fail or produce after some steps a substitution which instantiates some of the variables of G. Then, for a suitable substitution θ, all and-fair computations from $G\theta$ will fail. Therefore we can infer (by the NAF rule), $\neg G\theta$. The NAI rule is a variation of that: it allows to derive $\exists \neg G$, where $\exists G$ is the existential closure of G. In order to formulate the above in a rule we define $G_1 \simeq G_2$ if there exist substitutions θ_1, θ_2 such that $G_1\theta_1 = G_2$ and $G_2\theta_2 = G_1$.

$$\textbf{NAI rule} \quad \frac{\begin{array}{c}\text{all and-fair computations from } G \text{ either fail} \\ \text{or produce some substitution } \theta \text{ such that } G\theta \not\simeq G\end{array}}{\exists \neg G}.$$

In order to express this rule in our system, we have to make some assumption on the underlying constraint system, corresponding to the requirement of having infinitely many constant symbols $a_1, a_2, \ldots, a_i, \ldots$ in the language. Intuitively, the presence of those symbols is required so that we can always construct a substitution, for a finitely instantiating goal, in such a way that at some point some unification will fail. In a Herbrand constraint system, the presence of those constants leads to the presence of constraints of the form $x = a_1, x = a_2, \ldots, x = a_i, \ldots$ (for every variable x), which have the property of being inconsistent with every other constraint of the form $x = t$ with t a non-variable term, and their existential quantification with respect to x gives *true*. In a generic constraint system we can express this requirement in an abstract way as follows:

(ASS1) For every variable x there exist infinitely many constraints $d_{x0}, d_{x1}, \ldots, d_{xi}, \ldots$ such that

(i) for each $i \in \omega$, $\exists_x d_{xi} = \mathit{true}$, and

(ii) for each $i \in \omega$, for each $c \in \mathcal{C}$, if $\exists_x c \neq c$ and $c \not\sqsubseteq d_{xi}$, then $d_{xi} \sqcap c = \mathit{false}$.

Note that $\exists_x c \neq c$ is the abstract representation of the notion "x occurs free in c". The assumption (ASS1) could be made weaker by ruling out the diagonal elements from the requirement of inconsistency with the d_{xi}'s, provided that we assume, on the other hand, that the diagonal elements don't occur in programs and goals out of the context of the existential quantifier. We leave this extension for the reader to check.

Furthermore, we need other two assumptions:

(ASS2) For every constraint c, if $\exists_x c \neq c$, then for every constraint $d \neq \mathit{false}$ such that $d \sqsubseteq c$, $\exists_x d \neq d$ holds as well. This assumption means, intuitively, that if a variable occurs free in c, then it will remain free in any constraint obtained by adding information to c. The Herbrand constraint systems, for instance, satisfy this assumption.

(ASS3) $\exists_x \mathit{false} = \mathit{false}$.

(ASS4) Continuity of $\exists_x$: $\exists_x \sqcap_i d_i = \sqcap_i \exists_x d_i$, for every decreasing chain of constraints d_i.

From assumptions (ASS1) and (ASS3) it follows that all the d_{xi}'s are *isolated elements*, i.e. the only constraint they entail is *true*.

Lemma 46 *For every variable x, $i \in \omega$, $c \in \mathcal{C}$, if $d_{xi} \sqsubseteq c$ then either $c = d_{xi}$ or $c = \mathit{true}$.*

Proof Assume by contradiction that there exists $c \neq \mathit{true}, d_{xi}$, such that $d_{xi} \sqsubseteq c$. Thus $c \not\sqsubseteq d_{xi}$. By (ASS1) we have that $\exists_x d_{xi} = \mathit{true}$. Then by monotonicity of $\exists_x$ we have also $\exists_x c = \mathit{true}$. By (ASS1) we have then that $d_{xi} \sqcap c = \mathit{false}$, but, since $d_{xi} \sqsubseteq c$, this implies that $d_{xi} = \mathit{false}$. Therefore $\mathit{true} = \exists_x d_{xi} = \exists_x \mathit{false}$, which contradicts (ASS3). (We assume, of course, that $\mathit{true} \neq \mathit{false}$.)

We are now ready to reformulate the NAI rule. For the sake of simplicity, we restrict to the case of one variable x. The generalization of the NAI rule to the case of constraint logic programming is:

$$\frac{\begin{array}{l}\text{all and-fair computations from } G \text{ either fail}\\ \text{orproduce some constraint } c \text{ such that } \exists_x c \neq c\end{array}}{\exists_x \neg G}. \tag{5}$$

We can reformulate this rule in our framework as follows:

$$\frac{\mathcal{O}_P^\infty(\exists_x \neg G) = \mathit{True}}{\exists_x \neg G \longrightarrow \mathit{true}}. \tag{6}$$

The equivalence between (5) and (6) is shown by the following proposition:

Proposition 47 Rules (5) and (6) are equivalent, in the sense that the premise of (5) is satisfied iff the premise of (6) is satisfied.

Proof We need the following lemma:

Lemma 48 Let G be a positive goal. Assume that for all $c \in \mathcal{O}_P^\infty(G)$ such that $c \neq \mathit{false}$, $\exists_x c \neq c$ holds. Then there exists $i \in \omega$ such that for all $c \in \mathcal{O}_P^\infty(G)$, $d_{xi} \sqcap c = \mathit{false}$.

Proof Assume by contradiction that for all $i \in \omega$ there exists a $c_i \in \mathcal{O}_P^\infty(G)$ such that $d_{xi} \sqcap c_i \neq \mathit{false}$. Note that this implies that the set of constraints $\{c_i \mid d_{xi} \sqcap c_i \neq \mathit{false}\}$ is infinite since $d_{xi} \sqcap d_{xj} = \mathit{false}$, for $i \neq j$. Moreover, it follows by (**ASS1**) that $c_i \sqsubseteq d_{xi}$. It is not difficult to see that, if we enrich the transition system with transitions of the form $G \longrightarrow G$, for $G \in \mathit{TGoals}$, then we have

$$\mathcal{O}_P^\infty(G) = \bigcap_{j \in \omega} \bigcup \{\mathit{con}(G') \mid G \longrightarrow^j G'\},$$

where $\longrightarrow^j$ represents a sequence of transitions of length j. Define $C_j = \bigcup\{\mathit{con}(G') \mid G \longrightarrow^j G'\}$, for $j \in \omega$. Since $\mathit{con}(G')$ is finitary for every G' such that $G \longrightarrow^j G'$, and the transition relation is finitely branching we infer that C_j, for every $j \in \omega$, is finitary as well. We construct a decreasing chain of constraints $d_j \neq \mathit{false} \in C_j$ such that $\exists d_j = d_j$. This will give rise to a contradiction since $\sqcap_j d_j \neq \mathit{false} \in \mathcal{O}_P^\infty(G)$ (note that *false* is assumed to be a finite element) and $\exists_x \sqcap_j d_j = \sqcap_j \exists_x d_j = \sqcap_j d_j$ by (**ASS4**).

($j = 0$) Since $c_i \in C_0$, for every i, and C_0 is finitary, there exists a $d_0 \in C_0$ and a infinite set $I_0 \subseteq \omega$ such that $c_i \sqsubseteq d_0$, for all $i \in I_0$. We show that $\exists_x d_0 = d_0$: Assume that $\exists_x d_0 \neq d_0$. Then either $d_0 \sqsubseteq d_{xi}$ for all $i \in I_0$ or there exists a $i \in I_0$ such that $d_0 \not\sqsubseteq d_{xi}$. But since $d_{xi} \sqcap d_{xj} = \mathit{false}$ for $i \neq j$ the first case would imply that $d_0 = \mathit{false}$. On the other hand the second case implies that $d \sqcap d_{xi} = \mathit{false}$ for some d_{xi}. But $c_i \sqsubseteq d_0$ and $c_i \sqsubseteq d_{xi}$, so $c_i \sqsubseteq d_0 \sqcap d_{xi}$, which would imply that $c_i = \mathit{fasle}$.

($j > 0$) Again, like above, since C_j is finitary there exists a $d'_j \neq \mathit{false} \in C_j$ and a infinite set $I_j \subseteq I_{j-1}$ such that $c_i \sqsubseteq d'_j$, for all $i \in I_j$. Similarly as above we can prove that $\exists_x d'_j = d'_j$. Now define $d_j = d_{j-1} \sqcap d'_j$ and observe that $d_j \neq \mathit{false}$, and $\exists_x d_j = \exists_x(d_{j-1} \sqcap d'_j) = \exists_x d_{j-1} \sqcap \exists_x d'_j = d_{j-1} \sqcap d'_j = d_j$ (the second equation follows from Axiom (iii) of constraint systems).

We are now ready to prove Proposition 47.

($\Rightarrow$) Assume that all and-fair computation starting with G fail or produce some constraint c such that $\exists_x c \neq c$. It follows by (**ASS2**) that for all $c \in \mathcal{O}_P^\infty(G)$, such that $c \neq \mathit{false}$, $\exists_x c \neq c$ holds. By Lemma 48 we derive that there exists $i \in \omega$ such that $d_{xi} \in \neg\mathcal{O}_P^\infty(G)$. hence $\exists_x d_{xi} = \mathit{true} \in \boldsymbol{\exists}_x \neg\mathcal{O}_P^\infty(G) = \mathcal{O}_P^\infty(\exists_x \neg G)$.

($\Leftarrow$) Assume that $\mathit{true} \in \mathcal{O}_P^\infty(\exists_x \neg G) = \boldsymbol{\exists}_x \neg\mathcal{O}_P^\infty(G)$. Then there exists $d \in \neg\mathcal{O}_P^\infty(G)$ such that $\mathit{true} = \exists_x d$. For such d we have that $d \sqcap c = \mathit{false}$, for each $c \in \mathcal{O}_P^\infty(G)$. From this derive that $\exists_x \neq c$, for each $c \in \mathcal{O}_P^\infty(G)$, such that $c \neq \mathit{false}$. Since otherwise we would have that $\exists_x c = \exists_x c \sqcap \exists_x d = \exists_x(d \sqcap \exists_x c) = \exists_x(d \sqcap c) = \mathit{false}$ (by Axiom (iii) of constraint systems and (**ASS3**)), and thus $c = \mathit{false}$ (again by (**ASS3**)).

9 Declarative semantics for negation

In order to define a declarative interpretation of negation we introduce the following definition:

Definition 49 Let P be a program. For a positive goal G, we define the denotational semantics $\mathcal{G}^\infty$: $\mathcal{G}_P^\infty[\![G]\!] = \mathcal{G}[\![G]\!]\mathbf{fix}_\infty\mathcal{D}_P$, where $\mathbf{fix}_\infty\mathcal{D}_P$ denotes the greatest fixpoint of $\mathcal{D}_P$.

In [11] it is shown that $\mathcal{G}_P^\infty[\![G]\!]$, G positive, coincides with the notion of observables $\mathcal{O}_P^\infty(G)$ which includes the limit results of infinite computations. Observe that for terminal goals the semantics $\mathcal{G}_P^\infty$ coincides with $\mathcal{G}_P$. We extend $\mathcal{G}_P^\infty$ to negative goals: $\mathcal{G}_P^\infty[\![\neg G]\!] = \neg\mathcal{G}_P^\infty[\![G]\!]$. It follows that $\mathcal{G}_P^\infty[\![G]\!] = \mathcal{O}_P^\infty(G)$, for every goal.

We introduce implications of the form $c \Rightarrow \neg G$, for G a positive goal and define its meaning with respect to an environment as follows: $e \models c \Rightarrow \neg G$ if $\mathcal{G}[\![c]\!]e \subseteq \mathcal{G}[\![\neg G]\!]e$. We have the following declarative characterization of the observables of a negative goal:

Theorem 50 For every program P, constraint c, positive goal G we have that $P \models c \Rightarrow \neg G$ if and only if $c \in \mathcal{O}_P^\infty(\neg G)$.

Proof We have:

$$
\begin{array}{ll}
& c \in \mathcal{O}_P^\infty(\neg G) \\
\Leftrightarrow & \{\mathcal{G}_P[\![c]\!] = \downarrow c \text{ and } \mathcal{O}_P^\infty(\neg G) \text{ is downward-closed}\} \\
& \mathcal{G}_P[\![c]\!] \subseteq \mathcal{O}_P^\infty(\neg G) \\
\Leftrightarrow & \{\mathcal{O}_P^\infty(\neg G) = \mathcal{G}_P^\infty(\neg G) = \neg\mathcal{G}_P^\infty[\![G]\!]\} \\
& \mathcal{G}_P[\![c]\!] \subseteq \neg\mathcal{G}_P^\infty[\![G]\!] \\
\Leftrightarrow & \\
& \mathcal{G}_P[\![c]\!] \subseteq \neg\mathcal{G}[\![G]\!]\mathbf{fix}_\infty\mathcal{D}_P \\
\Leftrightarrow & \{(\Rightarrow) \text{ monotonicity of } \mathcal{G}[\![G]\!]\} \\
& \mathcal{G}[\![c]\!]e \subseteq \neg\mathcal{G}[\![G]\!]e, \text{ for every } e \text{ such that } e = \mathcal{D}_P(e) \\
\Leftrightarrow & \{\text{Theorem 26}\} \\
& \mathcal{G}[\![c]\!]e \subseteq \neg\mathcal{G}[\![G]\!]e, \text{ for every } e \text{ such that } e \models P \\
\Leftrightarrow & \\
& P \models c \Rightarrow \neg G.
\end{array}
$$

Note that $C_1 \subseteq C_2$ implies that $\neg C_2 \subseteq \neg C_1$, for any $C_1, C_2 \in \mathbf{C}$.

10 Conclusion and related work

We have reformulated the semantics of constraint logic programming in the framework of a more general language generated by a context-free grammar. We have defined the operational semantics via a transition system, following the SOS method, and, more precisely, the inductive approach to operational semantics. Denotationally, we have characterized clp goals (wrt a program) as elements of a distributive, cylindric lattice based on the underlying constraint system.

The denotational domain turned out to be a (cylindric) Heyting Algebra, which allows to express intuitionistic negation. We have compared this form of negation wrt negation as failure and negation as instantiation, and we have enriched the transition system with a rule for negation still maintaining the inductive style.

The notion of observables considered here corresponds to the so-called C-semantics ([12]), in which a program is regarded as a logical theory whose meaning is given by its *logical consequences*. This reflects on the fact that the observables are taken to be a downward-closed set of constraints. If the meaning of a program contains a certain constraint, then it also contains all the constraints entailed by it.

Other (more refined) notions of observables for clp have been investigateded in [15, 13, 14]. Of particular interest is the notion corresponding to the so-called S-semantics ([12]). This notion, in fact, is based on the idea of collecting only the exact outputs of a clp program, and therefore arises quite naturally when regarding clp as a standard programming language. In [8] we have shown that also the S-sematics of clp can be formulated in algebraic terms. The main difference is that the elements of the semantic domain are taken to be be arbitrary sets of constraints, instead of downward-closed ones. Also on this domain it is possible to define all the operators of the positive part of the language (hiding, conjunction etc.). What has not been investigated yet, and we plan it as a future work, is the algebraic definition of a suitable notion of negation in the context of S-semantics.

Acknowledgements

We would like to thank Daniele Micciancio for his helpful comments on a first draft of this paper.

References

[1] S. Abramsky. Domain Theory in Logical Form. *Proc. Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 47–53. IEEE, 1987. Extended version in Annals of Pure and Applied Logic, 51: 1–77, 1991.

[2] 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.

[3] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. Elsevier, 1977.

[4] E. Astesiano. Inductive semantics. In E.J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 51–136. Springer-Verlag, 1991.

[5] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 83–94. ACM, 1992.

[6] D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. International Conference on Logic Programming (ICLP)*, pages 111–125. The MIT Press, 1988.

[7] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[8] F. S. de Boer, A. Di Pierro, and C. Palamidessi. Nondeterminism and Infinite Computations in Constraint Programming. International Workshop on Topology and Completion in Semantics, Chartres, 1993. To appear in TCS.

[9] F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. In *Proc. Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 98–108. ACM, 1994.

[10] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation as Instantiation. *Information and Computation* 120(2), 263–278, 1995.

[11] A. Di Pierro and C. Palamidessi. A Logical Denotational Semantics for Constraint Logic Programming. In D. Sannella, editor, *Proc. European Symposium on Programming (ESOP)*, volume 788 of *Lecture Notes in Computer Science*, pages 195–21. Springer-Verlag, 1994.

[12] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[13] M. Gabbrielli. *The Semantics of Logic Programming as a Programming Language.* PhD Thesis, Technical Report TD–17/93, Department of Computer Science, University of Pisa, Pisa, 1993.

[14] M. Gabbrielli, G.M. Dore, and G. Levi. Observable Semantics for Constraint Logic Programming. *Journal of Logic and Computation*, 5(2):133–171,1995.

[15] M. Gabbrielli, and G. Levi. Modeling Answer Constraints in Constraint Logic Programming. In K. Furukawa, editor, *Proc. International Conference on Logic Programming (ICLP)*, pages 238–252. The MIT Press, Cambridge, Mass., 1991.

[16] 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.

[17] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I).* North-Holland, 1971.

[18] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 111–119. ACM, 1987.

[19] J. Maluszyński and T. Näslund. Fail Substitutions for Negation as Failure. In E. Lusk and R. Overbeck, editors, *Proc. North American Conference on Logic Programming (NACLP)*, pages 461–476. The MIT Press, 1989.

[20] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[21] D. Scott. Domains for denotational semantics. In M. Nielsen and E.M.Schmidt, editors, *Proc. International Colloquioum on Automata, Languages and Programming (ICALP)*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1994.

[22] E.Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983.

[23] J. C. Shepherdson. A sound and complete semantics for a version of negation as failure. *Theoretical Computer Science*, 65:343–371, 1989.

[24] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 232–245. ACM, 1990.

[25] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of Concurrent Constraint Programming. In *Proc. Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 333–353. ACM, 1991.

[26] 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.

[27] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of ACM*, 23(4):733–742, 1976.

[28] S. Vickers. *Topology via Logic*. Cambridge University Press, 1990.