A Logical Denotational Semantics for Constraint Logic Programming

Alessandra Di Pierro [*] and Catuscia Palamidessi [**]

Abstract. The process interpretation of constraint logic programming (clp) leads to a model which is similar for many aspects to (an unsynchronized version of) concurrent constraint programming (ccp). However, it differs from the latter because it supports the notion of consistency: an action can be performed only if it does not lead to an inconsistent store. We develop a denotational, fully abstract semantics for clp with respect to successful, failed and infinite observables. This semantics extends the standard model of clp in two ways: on one hand by capturing infinite computations; on the other hand by characterizing a more general notion of negation. Finally, our work can be regarded as a first step towards the development of a simple model for ccp with atomic tell.

1 Introduction

Constraint logic programming (clp, [10]) is an extension of logic programming ([28]) 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* ([27, 25]): 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, [23]).

However there are some important differences between clp and ccp. The latter cannot be regarded just as clp plus concurrency mechanisms. A central aspect of clp, in fact, is that inconsistent computations (i.e. computations which lead to an inconsistent result) are eliminated as soon as the inconsistency is detected. More precisely, the mechanism of choice in clp embodies a *consistency check*: a branch, whose first action would add a constraint inconsistent with the store, is disregarded. Such a check is not supported in ccp. On the other hand, the choice of ccp is controlled by an *entailment check*, which allows to enforce synchronization among processes. Such synchronization mechanism is not present in clp.

[*] 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@di.unipi.it

The concern for consistency in clp is reflected also by the notion of observables: usually a distinction is made between *success* (existence of a computation which leads to a consistent result) and *failure* (all fair computations lead to inconsistent results). This distinction is not made in ccp: *false*, the inconsistent store, is regarded as a result having the same "status" as the other constraints.

In this paper we present a logical model for clp which extends its standard semantics (hence it maintains the distinction between success and failure), and, on the other hand, it captures the notions related to processes, like the "results" of infinite computations. We first consider a structured operational semantics, based on a transition system, by means of which we define the notion of observables as the set of final constraints resulting from all possible derivations. Then we develop a compositional semantics, based on logical operators, and show its full correspondence with the observables. It turns out that it is more convenient to reason about a generalized language, which we call Γ_clp, structured as the free language generated by a BNF grammar. Γ_clp subsumes clp; the main extension is that it allows the presence of global variables in the clause bodies.

Moreover, we use this model to treat the problem of negative goals. In logic programming the meaning of a negative goal $\neg G$ is based on the *finite failure* of G ([6]). This works only for ground goals and does not allow to define a notion of computed result. More refined approaches consider a constructive notion: the result of $\neg G$ is, roughly, the negation of the disjunction of all the possible results of G ([5, 29, 26]), or it is obtained by finding "fail answers" ([17, 7]). Our approach has some points in common with the latter, but it is based on a different philosophy: we treat negation operationally as any other construct of the language, by means of structural rules. Only in the observables we use a different definition, which takes into account the non-monotonic nature of negative goals.

Since our model characterizes also infinite computations, it allows us to define two notions of negation. One corresponds to the standard one, and considers only the finite results. The other considers also the results of infinite computations, and it captures the set of constraints which, when added by an hypothetical interactive process at some stage of the computation, will cause the computation to fail.

Unfortunately, due to the non-monotonic nature of negation, it is not possible to introduce negative goals into the language, unless some restrictions are made. In fact this would cause the loss of the continuity of the semantic operator which is used in the fixpoint construction of the denotational semantics. In the literature of logic programming one can find various proposals to solve this problem. In particular we cite two approaches based on syntactical conditions: stratification ([2]) and strictness ([13]). These ideas generalize smoothly to our case.

1.1 Related work

The problem of characterizing the infinite computations via a fixpoint (denotational, bottom-up, declarative ...) semantics has been extensively studied in (constraint) logic programming. The main challenge is to get such a characterization while maintaining a simple domain of denotations.

In most concurrent languages, for instance the imperative languages and the languages with global nondeterminism, the denotational characterization of the operators requires complex structures, like synchronization trees or reactive sequences. On such domains there are well established techniques which allow to treat infinite computations, and they can be fruitfully applied also in the case of concurrent logic programming and concurrent constraint programming, see for instance [3], which is based on metric spaces. Another interesting approach, using category theory, is developed in [18].

In (constraint) logic programming, however, the domain of denotations for finite computations is particularly simple: sets of constraints or set of substitutions. Such a simple domain presents in principle more difficulties for treating infinite computations, because, for instance, it does not represent the occurrence of a computation step.

Most approaches aimed at modeling infinite computations with sets of constraints, are based on the greatest fixpoint of T_P, the immediate consequence operator which is used in logic programming for the fixpoint construction of the minimal model. Differences among these approaches depend on the kind of completion techniques applied on the underlying data structure, mainly based on partial orderings or metrics. However all these works have not been able to reach a full correspondence with the operational semantics. In the partial ordering completion of [8] only minimal answers are characterized. Furthermore the construction only works for clauses which contain at least one global variable. In the metric completion, at least in the approach found in the literature ([1, 14]) there is a basic soundness problem, because the objects which are considered are the solutions of the constraints rather than the constraint[3] themselves, and it might be the case that an infinite element is the solution of a constraint whereas its finite approximations aren't. Hence a limit element obtained in the fixpoint construction might be unobtainable operationally.

A different approach, based on adding to the program some suitable clauses containing indefinite terms, and then applying a least fixpoint construction, has been developed in [15]. However, also in this case completeness is not achieved.

In [11] infinite computations have been studied from a declarative point of view. However, the purpose of that work is not to characterize the results, but rather to establish a distinction between *infinite successes* and *infinite failures.* An infinite computation is "successful" whenever all partial results of the computation allow the same solution (hence the limit result has a solution). Otherwise it is considered an infinite failure. Infinite successes are shown to correspond to the difference set between the greatest and the least fixed points of T_P. The others are the difference set between $T_P \downarrow \omega$ and the greatest fixpoint of T_P. In our model the second difference set disappears, because we work on completed domains which ensure the downward continuity of T_P[4]. However also in our model a similar distinction between "successful" and "failed" infinite derivations can be made: infinite failures just correspond to the infinite computations delivering an inconsistent limit result.

The techniques we use in this paper have been inspired by the works in [12]

[3] The language investigated in [1, 14] is pure logic programming, hence constraints are equalities over the Herbrand universe, and solutions are syntactical unifiers.

[4] $T_P \downarrow \omega$ is the limit of the decreasing sequence $B, T_P(B), T_P^2(B), \ldots$ where B is the Herbrand base (the top element of the domain). It can be shown that $T_P \downarrow \omega$ is the complement of the set of finite failures.

and [16], which present a semantics for (angelic) ccp based on Scott-compact sets, capturing also infinite behavior. The ideas behind the denotational construction are quite similar; the difference is that we deal with a language supporting a notion of consistency check.

As far as we know, our approach to negation is quite original.

1.2 Plan of the paper

In next section we recall the definition of constraint system. Section 3 gives a brief description of clp and introduces the language Γ_clp. Section 4 illustrates the operational semantics of Γ_clp and the notion of observables. In Section 5 we develop the denotational model of Γ_clp and show its full correspondence with the observables. Finally, in Section 6 we enrich the language with negative goals, and we study a generalized notion of negation. Due to lack of space, we omit the proofs; they can be found in the full paper.

2 Constraint System

The concept of constraints over arbitrary domain is central for the paradigm of clp and represents its major novelty with respect to logic programming. We follow here the approach of [22], which defines the notion of constraint system along the lines of Scott's *information systems* [24].

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

In [22] a constraint systems is defined as an information system, but for the requirement of the consistency, which is removed. This is necessary in ccp in order to capture the possibility that a program gives rise to an inconsistent state during its execution. In our case this would not be necessary. However, we maintain this extension because this approach leads to constraint systems which are complete algebraic lattices. Constraint systems having this property are very desirable domains since their powerdomains can be algebraically characterized in terms of some family of sets instead of a family of sets of sets ([20]). In Section 5 we will use this property to define a simple denotational semantics for the language Γ_clp.

In this paper we regard a constraint system as a complete algebraic lattice in which the ordering $\sqsubseteq$ is the reverse of the the entailment relation $\vdash$, the top element *false* represents the inconsistent constraint, the bottom element *true* the empty constraint, and the lub operation $\sqcup$ the join of constraint, corresponding to the logical *and*. We refer to [22] for more details about the construction of such a structure.

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

An element $c \in C$ is *compact*, or *finite*, iff for any directed subset D of C, $c \sqsubseteq \bigsqcup D$ implies $c \sqsubseteq d$ for some $d \in D$. The lub of two finite elements is also finite.

Following the standard approach, we will sometimes use $\vdash$ instead of $\sqsubseteq$. Formally $c \vdash d \Leftrightarrow d \sqsubseteq c$.

2.1 Cylindric Constraint Systems

In order to model local variables in Γ_clp, a sort of hiding operator is needed. This can be formalized by introducing a kind of constraint system which supports *cylindrification operators*, a notion borrowed from the theory of cylindric algebras, due to Henkin, Monk and Tarski ([9]).

Assume given a (denumerable) set of variables Var with typical elements $x, y, z, \ldots$ and consider a family of operators $\{\exists_x \mid x \in Var\}$. Starting from a constraint system $\mathcal{C}$, construct a cylindric constraint system $\mathcal{C}'$ by taking $\mathcal{C}' = \mathcal{C} \cup \{\exists_x c \mid x \in Var,\ c \in \mathcal{C}'\}$ modulo the identities and with the additional relations derived by the following axioms:

(i) $\exists_x c \sqsubseteq c$,
(ii) if $c \sqsubseteq d$ then $\exists_x c \sqsubseteq \exists_x d$,
(iii) $\exists_x (c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$,
(iv) $\exists_x \exists_y c = \exists_y \exists_x c$,
(v) if $\{c_i\}_i$ is an increasing chain, then $\exists_x \bigsqcup_i c_i = \bigsqcup_i \exists_x c_i$.

Note that these laws force $\exists_x$ to behave as a first-order *existential operator*, as the notation suggests.

2.2 Diagonal constraint systems

In order to model parameter passing, it will be useful to enrich the constraint system with the so-called *diagonal constraints*, also from Henkin, Monk and Tarski ([9]).

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

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

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

3 The language

The syntax and the computational mechanism of clp ([10]) are very simple, but they do not provide a suitable basis to define a denotational semantics for clp. We need to reformulate the syntax of clauses and goals by means of a free grammar. The resulting language will be called Γ_clp. This will also allow us to describe the operational semantics in a structured way.

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

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.

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

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

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

$$p(x,a) \text{ :- } c(x), q(x,y)$$
$$p(b,x) \text{ :- }$$

can be rewritten in Γ_clp as

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

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

4 Operational semantics

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

We define the operational semantics in the style of SOS ([19]), i.e. by means of a transition system which describes the evolution of the network in a structural way. The configurations are goals, and the transition relation, $\longrightarrow$, represents the computation step.

4.1 The problem of the consistency check

In order to avoid the generation of goals with inconsistent constraints, we have to perform an appropriate consistency check. In Γ_clp it is more complicated than in clp, because of the generalized goals containing the $\vee$ construct. We have to define what is the constraint associated to such goals, and how does it combine with the constraint of a parallel agent[5]. 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 *greatest lower bound* $\sqcap$ of the two constraints of the disjuncts in the underlying constraint system. Unfortunately this choice does not work. Consider for example the constraint system illustrated in Figure 1. If we have the goal $x = 0 \vee x = 1$, then $x = 0 \sqcap x = 1 \equiv \mathit{true}$, which is consistent with the constraint $x = 2$. On the other hand, $(x = 0 \vee x = 1) \wedge x = 2$ should fail. The problem is that in order to correspond to the logical and and or, $\sqcup$ and $\sqcap$ should satisfy the distributive laws:

$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$$
$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c).$$

This is not the case in this example: the lattice in Figure 1 is not distributive.

A possible solution is to embed the constraint system into a distributive one, where the lub and glb model conjunction and disjunction[6]. A simple way to do this is to lift to sets of constraints, following the idea presented in [4]. In fact the set union and the set intersection, which are the glb and the lub on sets, enjoy the distributive property. It turns out that we actually need to consider only sets which

[5] An alternative would be to define the syntax and the operational semantics in such a way that disjunctions never occur in the goals. However this would complicate the transition system considerably.

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

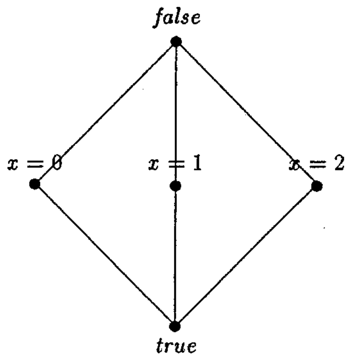

Fig. 1.: A non-distributive constraint system.

are *upward-closed* and *Scott-compact.* We recall that the upward closure of a set $D \subseteq \mathcal{C}$ is the set $\{c \in \mathcal{C} \mid \text{there exists } d \in D.\ d \sqsubseteq c\}$, which we denote by $\uparrow D$. A set $D \subseteq \mathcal{C}$ is upward-closed iff $D = \uparrow D$. Given a set $D \subseteq \mathcal{C}$, a *cover* for D is a set of compact elements C such that $D \subseteq \uparrow C$. Finally, a set $D \subseteq \mathcal{C}$ is Scott-compact iff every cover for D contains a finite subset which is also a cover. We will denote by $\mathcal{P}_{\text{cu}}(\mathcal{C})$ the set of Scott-compact upward-closed subsets of $\mathcal{C}$.

Given a cylindric, diagonal constraint system $(\mathcal{C}, \sqsubseteq, \sqcup, \mathit{true}, \mathit{false}, \mathit{Var}, \exists, d)$, consider the structure $\mathbf{C} = (\mathcal{P}_{\text{cu}}(\mathcal{C}), \supseteq, \cap, \mathit{True}, \mathit{False}, \mathit{Var}, \exists\!\!\exists, \delta)$, where *True* is $\uparrow \mathit{true}$, which coincides with $\mathcal{C}$, and *False* is $\uparrow \mathit{false}$, which coincides with $\{\mathit{false}\}$. For $C \in \mathcal{P}_{\text{cu}}(\mathcal{C})$, $\exists\!\!\exists_x C$ is defined as $\uparrow \exists_x C$, where $\exists_x C$ stands for the pointwise application of $\exists_x$ to the elements of C. Finally, δ_{xy} is defined as $\uparrow d_{xy}$. We have the following property:

Proposition 2. $\mathbf{C}$ *is a cylindric, diagonal constraint system with finite lub and glb coinciding with set intersection and set union respectively. Furthermore, if* $\{C_i\}_i$ *is an increasing chain (w.r.t.* $\supseteq$*) then* $lub_i C_i = \bigcap_i C_i$.

From the second part of this proposition it follows that

Corollary 3. *The support lattice of* $\mathbf{C}$ *is distributive.*

The correspondence between the original constraint system and $\mathbf{C}$ is obtained by mapping each element c into $\uparrow c$. It can be shown that this mapping preserves the ordering, the lub, and the existential operator. Of course, it does not preserve the glb.

The entailment relation on $\mathbf{C}$, i.e. the operational counterpart of the inverse of $\supseteq$, which we will denote by $\Vdash$, can be computed by extending the original entailment relation with the standard logical rules for conjunction and disjunction, plus the axioms for the existential operators and the diagonal elements ($\boldsymbol{\exists}_x$ and δ_{xy}) corresponding to the laws of Sections 2.1 and 2.2.

However, our main concern is proving the consistency of constraint C, i.e. that $C \neq \mathit{False}$, or, equivalently, that $C \not\Vdash \mathit{False}$ at least for those constraints C which belong to the set $\mathcal{L} \subseteq \mathcal{P}_{\mathrm{cu}}(\mathcal{C})$ generated by the following grammar:

$$C ::= \uparrow c \mid C \cap C \mid \boldsymbol{\exists}_x C \mid C \cup C.$$

To this purpose, having a complete deduction system for $\Vdash$ is not sufficient, because it does not imply the computability of $\not\Vdash$. Fortunately, in the case of the consistency check, we have a relatively complete system[7] which is described in Table 2.

The idea behind this system is to reduce the expressions to disjunctions of elementary expressions of the form $\uparrow c$, whose consistency we are able to test directly on the underlying constraint system. Note that to make this reduction we use the distributivity of $\cap$ wrt $\cup$ and the properties $\boldsymbol{\exists}_x \bigcup_i \uparrow c_i = \bigcup_i \uparrow \exists_x c_i$ and $\uparrow c \cap \uparrow d = \uparrow (c \sqcup d)$.

$$\frac{}{\uparrow c \Vdash \uparrow d}\ c \vdash d \qquad\qquad \frac{\bigcup_i \uparrow c_i \Vdash C \ ,\ \bigcup_j \uparrow d_j \Vdash D}{\bigcup_{ij} \uparrow (c_i \sqcup d_j) \Vdash C \cap D}$$

$$\frac{\bigcup_i \uparrow c_i \Vdash C}{\bigcup_i \uparrow \exists_x c_i \Vdash \boldsymbol{\exists}_x C} \qquad\qquad \frac{\bigcup_i \uparrow c_i \Vdash C \ ,\ \bigcup_j \uparrow d_j \Vdash D}{\bigcup_i \uparrow c_i \cup \bigcup_j \uparrow d_j \Vdash C \cup D}$$

$$\frac{}{\mathit{sat}(\uparrow c)}\ c \neq \mathit{false} \qquad\qquad \frac{\exists j : \mathit{sat}(\uparrow c_j) \ ,\ \bigcup_i \uparrow c_i \Vdash C}{\mathit{sat}(C)}$$

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

Proposition 4. **(Relative completeness of *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 $\mathit{sat}(C)$ iff $C \neq \mathit{False}$.*

Now we have the necessary machinery to define the operational semantics. Let's first define a function $\mathit{con} : \mathit{Goals} \to \mathcal{L}$, which gives the constraint established by a goal.

Definition 5.

$$\begin{aligned}
\mathit{con}(c) &= \uparrow c\\
\mathit{con}(p(x)) &= \uparrow \mathit{true}\\
\mathit{con}(G_1 \wedge G_2) &= \mathit{con}(G_1) \cap \mathit{con}(G_2)\\
\mathit{con}(G_1 \vee G_2) &= \mathit{con}(G_1) \cup \mathit{con}(G_2)\\
\mathit{con}(\exists_x G) &= \boldsymbol{\exists}_x \mathit{con}(G).
\end{aligned}$$

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

The consistency check on a goal G consists in verifying $sat(con(G))$.

4.2 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.

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

Table 3.: The transition system for Γ_clp.

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 [23], we express this link by the context $\exists_\alpha(d_{y\alpha} \wedge \exists_x(d_{\alpha x} \wedge \ldots))$, where α is a variable which does not occur in P. Note that through the whole computation only one variable α is needed. This mechanism for treating procedure calls is much simpler and more elegant than the machinery of standardization apart used in logic programming.

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 $con(G) \neq False$ implies $con(\exists_x G) \neq False$.

Parallel composition is modeled as interleaving.

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

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

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

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

4.3 The Observables

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

Definition 6. Given a program P, for every goal G we define

$$\begin{aligned}
\mathcal{O}_P(G) = \quad & \bigcup\{con(G') \mid G \longrightarrow^* G' \text{ for some } G' \in TGoals\} \\
& \cup \bigcup\{\uparrow false \mid \text{for every fair computation starting from } G, \\
& \qquad G \longrightarrow^* G' \not\longrightarrow, \text{ for some } G' \notin TGoals\} \\
& \cup \bigcup\{\textstyle\bigcap_n con(G_n) \mid \text{there exists an infinite fair computation} \\
& \qquad G_0 = G \longrightarrow G_1 \longrightarrow \ldots \longrightarrow G_n \longrightarrow \ldots\}.
\end{aligned}$$

Note that we consider a 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.

5 Denotational semantics

Our aim here is to give a denotational characterization 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 the semantic function will map goals into $\mathcal{P}_{\text{cu}}(\mathcal{C})$. The elements of this domain will be called *processes.*

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

Definition 7. Let *Pred* be a set of predicate symbols. Define

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

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

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

Proposition 8. *$(Env, \preceq)$ is a complete lattice.*

The equations defining the semantic interpretation functions for the denotational semantics are defined in Table 4.

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

Table 4.: The interpretation functions $\mathcal{D}$ and $\mathcal{G}$.

Proposition 9. *For every program P and goal G the functions $\mathcal{D}[\![P]\!] : Env \to Env$ and $\mathcal{G}[\![G]\!] : Env \to \mathcal{P}_{\text{cu}}(\mathcal{C})$ defined in Table 4 are continuous.*

We define the meaning $\mathcal{G}_P$ of a goal w.r.t. a program P, as

$$\mathcal{G}_P[\![G]\!] = \mathcal{G}[\![G]\!]\mathbf{fix}(\mathcal{D}[\![P]\!]),$$

where $\mathbf{fix}(\mathcal{D}[\![P]\!])$ is the least fixpoint of $\mathcal{D}$.

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

Theorem 10. *For every program P and goal G, $\mathcal{G}_P[\![G]\!] = \mathcal{O}_P(G)$ holds.*

6 A model for Negation

We consider now the possibility of introducing a construct for negation. We aim for the moment to have the possibility of computing negative goals, without using them in the bodies of the clause. Namely, in this work we restrict to *positive* programs.

Also we don't consider neither nested negation nor negation inside a $\vee$ context. So, our goals are conjunctions of positive and negative goals. In the following, G stands for a positive goal as defined in previous sections.

First of all we have to define what is the constraint associated to a negated goal. More in general, we have to extend the structure $\mathbf{C}$ with a notion of negation. The first idea would be to define $\neg C$ as $\mathcal{C} \setminus C$, but for doing this we should extend $\mathbf{C}$ with sets which are not upward closed. Another possibility is to define

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

This set is still upward closed, but might be not compact. It can be shown that the first notion of negation corresponds to classical negation, and the second one to intuitionistic negation. In this work we choose for the second, because it seems to combine better with the semantical construction developed so far. So, let's consider a structure $\mathbf{C}'$ which extends $\mathbf{C}$ in the sense that its support contains all the upward closed subsets of $\mathcal{C}$, not only the Scott-compact ones. In this structure, the set union and set intersection are the glb and the lub operators also with respect to infinite sets (of sets). Furthermore, also the infinitary distributive laws hold.

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

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

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

Example 1. Consider the program

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

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

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

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

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

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

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

6.1 Observing negation

As explained above, the negation rule reverts 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 11.

$$\begin{aligned}\mathcal{O}_P(\neg G) = \; & \bigcap\{con(\neg G') \mid \neg G \longrightarrow^* \neg G' \text{ for some } G' \in TGoals\} \\ & \cap \bigcap\{\uparrow true \mid \text{for every fair computation starting from } \neg G, \\ & \qquad \neg G \longrightarrow^* \neg G' \not\longrightarrow, \text{ for some } G' \notin TGoals\} \\ & \cap \bigcap\{\textstyle\bigcup_n con(\neg G_n) \mid \text{there exists an infinite fair computation} \\ & \qquad \neg G_0 = \neg G \longrightarrow \neg G_1 \longrightarrow \ldots \longrightarrow \neg G_n \longrightarrow \ldots\}.\end{aligned}$$

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

6.2 Denotation of negative goals

The denotational semantics $\mathcal{G}_P$ extends to negative goals as follows:

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

The correspondence with the observables is maintained:

Proposition 12. $\mathcal{G}_P[\![\neg G]\!] = \mathcal{O}_P(\neg G)$.

Note that this definition of negation has a close correspondence with the set of the *finite failures* of G as defined in [21]:

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

The difference is that in our case we have

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

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

7 Conclusions and future work

We have presented a generalized constraint logic programming with operators and, or, existential and (in a restricted form) negation. We have developed a structured operational semantics, embodying a mechanism for the appropriate consistency check. Then we have developed a natural model in which all the operators of the language have a logical meaning. This model is actually a complete *Heyting algebra*, and therefore it should allow to define a notion of intuitionistic implication. A future objective is to investigate this feature to see whether its counterpart in the language would have significance.

Another topic which seems to be interesting is the development of a more sophisticated transition system for negation which would allow to achieve a sort of constructive negation. This could be done by manipulating negative goals so to simplify them: For instance, $\neg(G_1 \vee G_2)$ should be transformed into $\neg G_1 \wedge \neg G_2$, and $\neg(G_1 \wedge G_2)$ should be transformed into $\neg G_1 \vee \neg G_2$. This would also make more efficient the system, because it allows to drag disjunctions out of negation and apply the disjunction rule. Another advantage is that in this way we don't need anymore to define a different notion of observables: the negative goals will in fact be completely reduced, at the end of a computation, to conjunctions of constraints. Therefore, we would have a structural semantics even when negation is a free constructor in the goals, thus generating nested negations, conjunctions of negations etc.

Another problem to investigate is how to allow negations in the bodies of the clauses. As explained in the introduction, this would compromise the monotonicity and the continuity of the semantic operator $\mathcal{D}$, which therefore would not be guaranteed to have a least fixpoint. So, either we put some restrictions on the clauses with negation, like stratification ([2]) or and strictness ([13]), or we find another way (some appropriate fixpoint) to characterize the intended meaning of a program.

References

1. M.A. Nait Abdallah. On the intepretation of infinite computations in logic programming. In J. Paredaens, editor, *Proc. of Automata, Languages and Programming*, volume 172, pages 374–381. Springer Verlag, 1984.
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. F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. Semantic models for Concurrent Logic Languages. *Theoretical Computer Science*, 86(1), 3–33, 1991.
4. F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, 1993.
5. D. Chan. Constructive Negation Based on the Completed Database. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 111–125. The MIT Press, 1988.
6. K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
7. W. Drabent. Constructive Negation by Fail Answers. In *Proc. of the Workshop on Logic Programming and Non-monotonic reasoning*, 1993. To appear.
8. W.G. Golson. Toward a declarative semantics for infinite objects in logic programming. *Journal of Logic Programming*, 5:151–164, 1988.
9. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I)*. North-Holland, 1971.
10. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

11. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Department of Computer Science, Monash University, June 1986.
12. R. Jagadeesan, V.A. Saraswat, and V. Shanbhogue. Angelic non-determinism in concurrent constraint programming. Technical report, Xerox Park, 1991.
13. K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7(3):231–245, 1989.
14. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987. Second edition.
15. G. Levi and C. Palamidessi. Contributions to the semantics of logic perpetual processes. *Acta Informatica*, 25(6):691–711, 1988.
16. M. Z. Kwiatkowska. Infinite Behaviour and Fairness in Concurrent Constraint Programming. In J. W. de Bakker, W. P.de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 348–383, Beekbergen The Nederland, June 1992. REX Workshop, Springer-Verlag, Berlin.
17. J. Maluszyński and T. Näslund. Fail Substitutions for Negation as Failure. In E. Lusk and R. Overbeck, editors, *Proc. North American Conf. on Logic Programming'89*, pages 461–476. The MIT Press, 1989.
18. S. Nystrom and B. Jonsson. In D. Miller, editor, *Proc. International Symposium on Logic Programming'93*, pages 335–352. The MIT Press, 1993.
19. G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
20. G. Plotkin. Domains. Department of Computer Science, University of Edimburgh, 1992. Post-graduate lecture notes in advanced domain theory (incorporating the 'Pisa notes' 1981).
21. J. C. Shepherdson. A sound and complete semantics for a version of negation as failure. *Theoretical Computer Science*, 65:343–371, 1989.
22. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. Seventeenth Annual ACM Symp. on Principles of Programming Languages*, pages 232–245, New York, 1990.
23. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*, New York, 1991.
24. D. Scott. Domains for denotational semantics. In *Proc. of ICALP*, 1982.
25. E.Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983.
26. P. Stuckey. Constructive negation for constraint logic programming. In *Proc. sixth Annual Symposium on Logic in Computer Science*, 1991.
27. 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.
28. 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.
29. M. G. Wallace. Negation by Constraints: a Sound and Efficient Implementation of Negation in Deductive Databases. In *IEEE Int'l Symp. on Logic Programming*, pages 253–263. IEEE, 1987.