Operational Semantics of Constraint Logic Programming over Finite Domains

Pascal Van Hentenryck[1]
Yves Deville[2]

Technical Report No. CS-90-23
October 1990

[1]Computer Science Dept., Brown University, Providence, RI 02912
[2]University of Namur, 21 rue Grandgagnage, B-5000 Namur, Belgium

Operational Semantics of Constraint Logic Programming over Finite Domains

Pascal Van Hentenryck
Brown University, Box 1910,
Providence, RI 02912
Email: pvh@cs.brown.edu

Yves Deville
University of Namur, 21 rue Grandgagnage
B-5000 Namur (Belgium)
Email: yde@info.fundp.ac.be

Abstract

Although the Constraint Logic Programming (CLP) theory [7] is an elegant generalization of the LP theory, it has some difficulties in capturing some operational aspects of actual CLP languages (e.g. [8, 6, 3, 12, 18, 19]). A difficulty comes from the intentional incompleteness of some constraint-solvers. Some constraints are simply delayed until they can be decided by the constraint-solver. Others are approximated, providing an active pruning of the search space without being actually decided by the constraint-solver.

This paper presents an extension of the *Ask & Tell* framework [14] in order to give a simple and precise operational semantics to (a class of) CLP languages with an incomplete constraint-solver. The basic idea is to identify a subset of the constraints (the basic constraints) for which there exists an efficient and complete constraint-solver. Non-basic constraints are handled through two new combinators, relaxed ask and relaxed tell, that are in fact relaxations of the standard ask and tell.

The extended framework is instantiated to CLP on finite domains, say CLP(F) [16, 6]. Arc-consistency is shown to be an efficient and complete constraint-solver for basic constraints. We also present how non-basic constraints can be approximated in CLP(F). The resulting semantics precisely describes the operational semantics of the language, enables the programmer to reason easily about the correctness and efficiency of his programs, and clarifies the links of CLP(F) with the CLP and *Ask & Tell* theories.

It is believed that the approach can be used as well to endow other CLP languages such as BNR-Prolog [12], CLP(Σ^*) [19], and parts of Trilogy [18] with a precise operational semantics.

1 Introduction

Constraint Logic Programming (CLP) is a generalization of Logic Programming (LP) where unification, the basic operation of LP languages, is replaced by the more general concept of

constraint-solving over a computation domain.

Syntactically, a CLP program can be seen as a finite set of clauses of the form

$$H \leftarrow c_1 \wedge \ldots \wedge c_n \diamond B_1 \wedge \ldots \wedge B_m$$

where $H, B_1, \ldots, B_m$ are atoms and $c_1, \ldots, c_n$ are constraints. A goal in a CLP language is simply a clause without head.

The declarative semantics of a CLP language can be defined either in terms of logical consequences or in an algebraic way. An answer to a CLP goal is no longer a substitution but rather a conjunction of constraints $c_1, \ldots, c_n$ such that

$$P, \mathcal{T} \models (\forall)(c_1 \wedge \ldots \wedge c_n \Rightarrow G) \text{ (logical version)}$$
$$P \models_{\mathcal{S}} (\forall)(c_1 \wedge \ldots \wedge c_n \Rightarrow G) \text{ (algebraic version)}$$

where P is a program, $\mathcal{S}$ is a structure, $\mathcal{T}$ is the theory axiomatizing $\mathcal{S}$, and $(\forall)(F)$ represents the universal closure of F. The rest of the presentation can be read from a logic or algebraic point of view and we use the notation $\mathcal{D} \models$ to denote that fact.

The operational semantics of CLP amounts to replacing unification by constraint-solving. It might be defined by considering configurations of the form $\langle G, \sigma \rangle$ where G is a conjunction of atoms and σ is a consistent conjunction of constraints. Now the only transition that needs to be defined between configurations is the following:

$$\frac{H \leftarrow c_1 \wedge \ldots \wedge c_n \diamond B_1 \wedge \ldots \wedge B_m \in P \qquad \mathcal{D} \models (\exists)(\sigma \wedge c_1 \wedge \ldots \wedge c_n \wedge H = A)}{\langle A \wedge G, \sigma \rangle \longmapsto \langle B_1 \wedge \ldots \wedge B_m \wedge G, \sigma \wedge c_1 \wedge \ldots \wedge c_n \wedge H = A \rangle}$$

In the above transition rule, A is the selected atom (it can be any atom in the goal since the order in a conjunction is irrelevant) and $(\exists)(F)$ represents the existential closure of F.

The CLP theory [7] imposes a number of restrictions on $\mathcal{S}$, $\mathcal{T}$, and their relationships to establish equivalences between the semantics. Also the CLP language should be embedded with a *complete* constraint-solver, which means that, given a conjunction of constraints σ, the constraint-solver should return *true* if $\mathcal{D} \models (\exists)(\sigma)$ and $false$ otherwise (i.e $\mathcal{D} \models (\forall)(\neg\sigma)$).

Although the CLP theory is an elegant generalization of the LP theory, it has some difficulties in capturing all operational aspects of actual CLP languages such as [8, 6, 3, 12, 18, 19]. One difficulty comes from the possibility offered by these languages to state constraints that cannot be decided upon by the constraint-solver. This is the case for instance of non-linear constraints over rational and real numbers which are simply delayed until they become linear. Another difficulty comes from the fact that, for some classes of problems, an intentionally incomplete but efficient constraint-solver might well turn out to be more valuable from a programming standpoint than a complete constraint-solver. This is justified by the trade-off, that appears in many combinatorial problems, between the time spent in pruning and searching.

The present paper originates from an attempt to define in a precise and simple way the operational semantics of CLP over finite domains, say CLP(F) [16, 6], which suffers for both difficulties mentioned above. CLP(F) was designed with the goal of tackling a large class of combinatorial optimization problems with a short development time and an efficiency competitive with procedural languages. Typical applications for which CLP(F) is successful include sequencing and scheduling, graph coloring and time-table scheduling, as well as various assignments problems. The basic idea behind CLP(F) is to associate to variables a domain which is a finite set of values. Constraints in CLP(F) include (possibly non-linear) equations, inequalities, and disequations. It is simple to see that a complete constraint-solver exists for the above constraints but would necessarily require exponential time (unless P = NP). Moreover many of the abovementioned applications require different solutions and use various kinds of constraints, heuristics, and problem features. Hence it is unlikely that a complete constraint-solver be adequate for all of them. Fortunately most of them basically share the same pruning techniques and the idea behind CLP(F) was precisely to provide the language with those techniques[1]. However CLP(F) does not inherit directly its operational semantics from the CLP framework since, on the one hand, some constraints are only used when certain conditions are satisfied (e.g. disequations) and, on the other hand, some constraints are used to prune the search space although the constraint-solver cannot, in general, decide their satisfiability (e.g. inequalities). Previous approaches to define the operational semantics of CLP(F) were not based on the CLP framework but rather were given in terms of inference rules [15].

The new operational semantics presented here is based on the *Ask & Tell* framework [14] which generalizes the CLP framework by adding the concept of constraint entailment (i.e. ask) to the concept of constraint-solving (i.e. tell). Ask constraints directly account for the first difficulty in CLP languages: they might be used to restrict the context in which a constraint is executed. However to account for the incompleteness of the constraint-solver, we need to generalize the framework[2]. The basic idea behind the semantics presented here is to split the set of primitive constraints into two sets:

- *basic* constraints for which there exists an efficient and complete constraint-solver;
- *non-basic* constraints which are only approximated.

The basic constraints are handled following the standard CLP theory and are precisely those constraints that can be returned as an answer to a goal. Non-basic constraints are handled through two new combinators, relaxed ask and relaxed tell. Contrary to ask (resp. tell), relaxed ask (resp. relaxed tell) check entailment (resp. consistency) not wrt the accumulated constraints but rather wrt a relaxation of them. Moreover relaxed tell enables to deduce new basic constraints approximating the non-basic one. Formally, relaxed ask

[1]Note that for other types of problems a complete constraint-solver might be more appropriate [1].

[2]In fact Saraswat considers his framework as parametrized on the combinators as well so that we are actually instantiating his framework.

and relaxed tell are not mere combinators but rather they define a family of combinators parametrized by a relaxation and an approximation function. The relaxation function specifies the relaxation of the accumulated constraints while the approximation specifies how to infer new basic constraints.

Describing a CLP language in the extended framework (i.e. with relaxed ask and tell) amounts to specifying:

- the basic constraints and their associated complete constraint-solver;
- the non-basic constraints and their associated (1) relaxation functions, (2) approximation functions, and (3) constraint-solvers that are complete for the conjunction of a non-basic constraint and the relaxation of a conjunction of basic constraints.

The extended framework is then instantiated to CLP(F). We identify the subset of basic constraints and show that arc-consistency [9] provides an efficient and complete constraint-solver for them. We also define the relaxation and approximation functions used in CLP(F) and suggest the non-basic constraint-solvers.

The contributions of the paper are twofold.

1. It gives a precise and simple operational semantics to CLP(F), characterizing what is computed (e.g. what is an answer) and how the computation is achieved (e.g. what is the pruning at some computation point). The semantics allows the programmer to formally reason about the correctness and the efficiency of his programs. It also clarifies the relationships between CLP(F) on the one hand, and the CLP and *Ask&Tell* frameworks on the other hand.
2. It proposes an extended framework that can possibly be instantiated to endow languages such as BNR-Prolog [12], CLP(Σ^*) [19], and parts of Trilogy [18] with a precise operational semantics.

The rest of this paper is organized in the following way. The next section defines the operational semantics of basic CLP. Section 3 instantiates basic CLP to finite domains. Section 4 describes the new combinators for non-basic constraints, relaxed ask and relaxed tell. Section 5 instantiates relaxed ask and relaxed tell to CLP(F). Section 6 contains the conclusion of this paper and directions for future research.

2 Basic CLP

In this section, we define the syntax and operational semantics of the class of languages we consider for Constraint Logic Programming. We use a structural operational semantics [13] similar to the one used in [14]. There is an important difference however due to the various possible interpretations of nondeterminism. Saraswat's semantics, as well as other semantics for concurrent logic programming languages, describes all possible executions of a program

on a given goal. Hence an actual implementation might actually fail (resp. succeed) for some elements of the success (resp. failure) set. Our semantics captures a single execution and hence an element of the success set might never fail in an actual implementation. This difference does not show up in the transition rules but rather in the definition of the success, failure, flounder, and divergence sets.

2.1 Syntax

The abstract syntax of the basic language can be defined by the following (incomplete) grammar:

$$P ::= H \leftarrow B \mid P.P$$
$$B ::= A \mid ask(c) \rightarrow B \mid tell(c) \mid B \,\&\, B \mid \exists x\, B \mid true$$

In words, a program is a non-empty set of clauses. A clause is composed of a head and a body. The head is an atom whose arguments are all distinct variables. The body is either an atom, an implication $ask(c) \rightarrow B$ where $ask(c)$ is an ask on constraint c, a tell on constraint c, a conjunction of two bodies, an existential construction $\exists x\, B$ where B is a body with variable x free, or *true*. For completeness, some semantic rules should also be added, for instance the rule stating that any variable in a clause is either existentially quantified or appears in the head.

The concrete syntax can be the one of any existing CLP language suitably enhanced to include the implication construct. There is no difficulty in translating any of these concrete syntax to the abstract one.

2.2 Basic Constraints

Basic constraints are split into two sets: basic tell-constraints, simply referred to as basic constraints, and basic ask-constraints. The constraint-solver for basic CLP should be complete for consistency of basic constraints and entailment of basic ask-constraints.

Definition 1 A basic constraint-solver is *complete* iff it can decide

1. consistency of c and σ (i.e. $\mathcal{D} \models (\exists)(\sigma \wedge c)$);
2. entailment of c' wrt σ (i.e. $\mathcal{D} \models (\forall)(\sigma \Rightarrow c')$);

where c is a basic constraint, c' a basic ask-constraint, and σ a conjunction of basic constraints.

CLP languages usually maintain constraints in a reduced form to obtain an incremental behaviour necessary to achieve efficiency[3].

[3]Most operational semantics do not include this function as it plays no fundamental role in their description. As we will see, it plays an important role in showing how efficiently the relaxation function can be computed in CLP(F).

Definition 2 A *reduction* function is a total function red which, given a consistent conjunction of basic constraints σ, returns a conjunction of basic constraints, such that $\mathcal{D} \models \sigma \Leftrightarrow red(\sigma)$.

In the following, we assume the existence of a reduction function red for basic constraints and denote by CCR the set of consistent conjunctions of constraints in reduced form (i.e. the codomain of red). The actual choice of the reduction function depends upon the computation domain.

2.3 Structural Operational Semantics

2.3.1 Configurations

The configurations in the transition system are of the form $\langle B, \sigma \rangle$ where B is a body and σ a conjunction of basic constraints in reduced form. Informally B represents what remains to be executed while σ represents the constraints accumulated so far. Successful computations end up with a conjunction of basic constraints in reduced form. Hence $CCR \subseteq T$. Computations may also flounder when an ask on constraint c cannot be decided upon (i.e. neither c nor $\neg c$ is entailed by the accumulated constraints). We use the terminal $flounder$ to capture that behaviour.
Terminal configurations are thus described by

$$T \;=\; \{\sigma \mid \sigma \in CCR\} \cup \{flounder\}.$$

Configurations are described by

$$\Gamma \;=\; \{\langle B, \sigma \rangle \mid B \text{ is a body and } \sigma \in CCR\} \cup T.$$

A transition $\gamma \longmapsto \gamma'$ can be read as "configuration γ nondeterministically reduces to γ'".

2.3.2 Transition Rules

Goals are assumed to be resolved against clauses from some program and constraints are assumed to be checked for consistency and entailment in a given structure or theory $\mathcal{D}$. Since the structure (or theory) never changes, we simply omit it and assume that it is clear from the context. As the program changes, we make use of an indexed transition system and use $P \vdash \gamma \longmapsto \gamma'$ to denote that the transition $\gamma \longmapsto \gamma'$ takes place in the context of program P. When the transition does not depend on program P, we simply drop the prefix $P \vdash$. In the following, σ denotes a conjunction of basic constraints in reduced form, γ a configuration, B and G bodies, P a program, and c a basic constraint. These are possibly subscripted.

True: The atom *true* simply leads to a terminal configuration.

$$\langle true \;,\; \sigma \rangle \longmapsto \sigma$$

Tell: The tell operation on a basic constraint is successful if the constraint is consistent with the accumulated constraints.

$$\frac{\mathcal{D} \models (\exists)(\sigma \wedge c)}{\langle tell(c)\,,\,\sigma\rangle \longmapsto red(\sigma \wedge c)}$$

Implication: An implication $ask(c) \rightarrow B$ never fails. If the basic ask-constraint c is entailed by the accumulated constraints, it reduces to the body B. If $\neg c$ is entailed by the accumulated constraints, the implication terminates successfully. Otherwise, the implication flounders.

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow c)}{\langle ask(c) \rightarrow G\,,\,\sigma\rangle \longmapsto \langle G, \sigma\rangle}$$

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow \neg c)}{\langle ask(c) \rightarrow G\,,\,\sigma\rangle \longmapsto \sigma}$$

$$\frac{\begin{array}{l}\mathcal{D} \models \neg(\forall)(\sigma \Rightarrow c)\\ \mathcal{D} \models \neg(\forall)(\sigma \Rightarrow \neg c)\end{array}}{\langle ask(c) \rightarrow G\,,\,\sigma\rangle \longmapsto flounder}$$

Existential Quantification: An existential quantification can be removed by replacing the quantified variable by a brand new variable.

$$\frac{\begin{array}{l}y \text{ is a brand new variable}\\ \langle G[x/y]\,,\,\sigma\rangle \longmapsto \gamma\end{array}}{\langle \exists x\, G\,,\,\sigma\rangle \longmapsto \gamma}$$

The fact that variable y be brand new can be formalized in various ways if necessary.

Conjunction: The semantics of conjunction is given here by the interleaving rule which is appropriate for CLP languages. If any of the goals in a conjunction can make a transition, the whole conjunction can make a transition as well and the accumulated constraints are updated accordingly. The conjunction flounders when both conjuncts flounder.

$$\frac{\langle G_1\,,\,\sigma\rangle \longmapsto \langle G_1'\,,\,\sigma'\rangle}{\begin{array}{l}\langle G_1 \,\&\, G_2\,,\,\sigma\rangle \longmapsto \langle G_1' \,\&\, G_2\,,\,\sigma'\rangle\\ \langle G_2 \,\&\, G_1\,,\,\sigma\rangle \longmapsto \langle G_2 \,\&\, G_1'\,,\,\sigma'\rangle\end{array}}$$

$$\frac{\langle G_1\,,\,\sigma\rangle \longmapsto \sigma'}{\begin{array}{l}\langle G_1 \,\&\, G_2\,,\,\sigma\rangle \longmapsto \langle G_2\,,\,\sigma'\rangle\\ \langle G_2 \,\&\, G_1\,,\,\sigma\rangle \longmapsto \langle G_2\,,\,\sigma'\rangle\end{array}}$$

$$\frac{\langle G_1\ ,\ \sigma\rangle \longmapsto \mathit{flounder} \quad \langle G_2\ ,\ \sigma\rangle \longmapsto \mathit{flounder}}{\langle G_1 \,\&\, G_2\ ,\ \sigma\rangle \longmapsto \mathit{flounder}}$$

Procedure Call with one Clause: We now consider the solving of an atom $p(t_1, \ldots, t_n)$ wrt a clause $p(x_1, \ldots, x_n) \leftarrow B$. If $B[x_1/t_1, \ldots, x_n/t_n]$ can make a transition in the context of the accumulated constraints, then so can $p(t_1, \ldots, t_n)$ in the context of the clause and the constraints.

$$\frac{p(x_1, \ldots, x_n) \leftarrow B \ \vdash\ \langle B[x_1/t_1, \ldots, x_n/t_n]\ ,\ \sigma\rangle \longmapsto \gamma}{p(x_1, \ldots, x_n) \leftarrow B \ \vdash\ \langle p(t_1, \ldots, t_n)\ ,\ \sigma\rangle \longmapsto \gamma}$$

Procedure Call with several Clauses: If an atom can make a transition in program P_1, it can obviously make a transition in the program made up of P_1 and program P_2.

$$\frac{P_1 \ \vdash\ \gamma \longmapsto \gamma'}{\begin{array}{l} P_1.P_2 \ \vdash\ \gamma \longmapsto \gamma' \\ P_2.P_1 \ \vdash\ \gamma \longmapsto \gamma' \end{array}}$$

2.4 Operational Semantics

We now define the operational semantics of the language in terms of its success, floundering, divergence, and failure sets. Let $\stackrel{*}{\longmapsto}$ denote the reflexive and transitive closure of $\longmapsto$ and $\mathit{initial}(G, \sigma)$ the configuration $\langle G, \mathit{red}(\sigma)\rangle$. Also a configuration γ is said to diverge in program P if there exists an infinite sequence of transitions

$$P \vdash \gamma \longmapsto \gamma_1 \longmapsto \ldots \longmapsto \gamma_i \longmapsto \ldots$$

The success, floundering, and divergence sets (not necessarily disjoint) can be defined in the following way.

$$\begin{array}{lcl} SS[P] & = & \{\langle G, \sigma\rangle \mid P \vdash \mathit{initial}(G, \sigma) \stackrel{*}{\longmapsto} \sigma'\} \\ FLS[P] & = & \{\langle G, \sigma\rangle \mid P \vdash \mathit{initial}(G, \sigma) \stackrel{*}{\longmapsto} \mathit{flounder}\} \\ DS[P] & = & \{\langle G, \sigma\rangle \mid \mathit{initial}(G, \sigma) \text{ diverges in P}\} \end{array}$$

The failure set can now be defined in terms of the above three sets.

$$FS[P] \ = \ \{\langle G, \sigma\rangle \mid \langle G, \sigma\rangle \notin SS[P] \cup FLS[P] \cup DS[P]\}$$

Another semantic definition can be given to capture the results of the computation.

$$RES[P]\langle G, \sigma\rangle \ = \ \{\sigma' \mid P \vdash \mathit{initial}(G, \sigma) \stackrel{*}{\longmapsto} \sigma'\}$$

3 Basic CLP(F)

In this section, we instantiate basic CLP to finite domains.

3.1 Basic Constraints

In the following, x and y, possibly subscripted, denote variables and a, b, c, v, w, possibly subscripted, denote natural numbers.

Definition 3 The *basic* constraints of CLP(F) are either *domain* constraints or *arithmetic* constraints.

- domain constraint: $x \in \{v_1, \ldots, v_n\}$;
- arithmetic constraints:
  - $ax \neq b$;
  - $ax = b$;
  - $ax = by + c$ $(a \neq 0 \neq b)$;
  - $ax \geq by + c$;
  - $ax \leq by + c$;

The semantics of addition, multiplication, $=$, $\leq$, $\geq$, and $\neq$ is the usual one. Clearly, the negation of each basic constraint can be expressed as a conjunction or disjunction of basic constraints. Hence all the basic constraints can also be basic ask-constraints[4]. Note that the variables appearing in arithmetic constraints are expected to appear in some domain constraints. Every variable thus has a domain. This can be seen as an implicit ask-constraint and justifies the following definition.

Definition 4 A *system of constraints* S is a pair $\langle AC, DC \rangle$ where AC is a set of arithmetic constraints and DC is a set of domain constraints such that any variable occurring in an arithmetic constraint also occurs in some domain constraint of S.

Definition 5 Let $S = \langle AC, DC \rangle$ be a system of constraints. The set D_x is the *domain* of x in S (or in DC) iff the domain constraints of x in DC are $x \in D_1, \ldots, x \in D_k$ and D_x is the intersection of the D_i's.

It follows that, provided that each variable has a domain, a conjunction of basic constraints can be represented by a system of constraints and vice versa.

We conclude this subsection by a number of conventions. If c is an arithmetic constraint with only one variable x, we say that c is *unary* and denote it as $c(x)$. Similarly, if c is an arithmetic constraint with two variables x and y, we say that c is *binary* and denote it $c(x, y)$. As usual, $c(x/v)$ and $c(x/v, y/w)$ denote the Boolean value obtained from $c(x)$ and $c(x, y)$ by replacing x and y by the values v and w respectively.

[4]Note that we may want to consider some of them as non-basic constraints for efficiency reasons.

3.2 Constraint-Solving

The constraint-solver of CLP(F), and hence of basic CLP(F), is based on consistency techniques, a paradigm emerging from AI research [9]. We start by defining a number of notions.

Definition 6 Let $c(x)$ be a unary constraint and D_x be the domain of x. Constraint $c(x)$ is said to be *node-consistent wrt* D_x if $c(x/v)$ holds for each value $v \in D_x$.

Definition 7 Let $c(x,y)$ be a binary constraint and D_x, D_y be the domains of x, y. Constraint $c(x,y)$ is said to be *arc-consistent wrt* D_x, D_y if the following conditions hold:

1. $\forall v \in D_x \ \exists w \in D_y \ c(x/v, y/w)$ holds;
2. $\forall w \in D_y \ \exists v \in D_x \ c(x/v, y/w)$ holds;

We are in position to define a solved form for the constraints.

Definition 8 Let S be a system of constraints. S is in *solved form* iff any unary constraint $c(x)$ in S is node-consistent wrt the domain of x in S, and any binary constraint $c(x,y)$ in S is arc-consistent wrt the domains of x, y in S.

We now study a number of properties of systems of constraints in solved form.

Property 9 Let $c(x,y)$ be the binary constraint $ax \geq by + c$, arc-consistent wrt $D_x = \{v_1, \ldots, v_n\}, D_y = \{w_1, \ldots, w_m\}$. Assume also that $v_1 < \ldots < v_n$ and $w_1 < \ldots < w_m$. Then we have

1. $c(v_1, w_1)$ and $c(v_n, w_m)$ hold;
2. if $c(v_i, w_j)$ holds, then $c(v_{i+k}, w_{j-l})$ holds $(0 \leq k \leq n-i, \ 0 \leq l < j)$.

Proof

- (2): Since $v_{i+k} \geq v_i$, $w_j \geq w_{j-l}$ and a, b are natural numbers, we have that $av_{i+k} \geq av_i \geq bw_j + c \geq bw_{j-l} + c$. Hence $c(v_{i+k}, w_{j-l})$ holds.
- (1): By arc-consistency, $c(v_1, w)$ and $c(v, w_n)$ holds for some $w \in D_y$ and some $v \in D_x$.Hence, by (1), $c(v_1, w_1)$ and $c(v_n, w_m)$ hold.

□

Property 10 Let $c(x,y)$ be the binary constraint $ax \leq by + c$, arc-consistent wrt $D_x = \{v_1, \ldots, v_n\}, D_y = \{w_1, \ldots, w_m\}$. Assume also that $v_1 < \ldots < v_n$ and $w_1 < \ldots < w_m$. Then we have

1. $c(v_1, w_1)$ and $c(v_n, w_m)$ hold;

2. if $c(v_i, w_j)$ holds, then $c(v_{i-k}, w_{j+l})$ holds $(0 \leq k < i,\ 0 \leq l \leq m - j)$.

Property 11 Let $c(x, y)$ be the binary constraint $ax = by + c$, arc-consistent wrt $D_x = \{v_1, \ldots, v_n\}, D_y = \{w_1, \ldots, w_m\}$. Assume also that $v_1 < \ldots < v_n$ and $w_1 < \ldots < w_m$. Then we have $n = m$ and $c(v_i, w_i)$ holds $(1 \leq i \leq n)$.

Proof The proof is by induction on i. Applying Properties 1 and 2 to the inequalities associated with $c(x, y)$ implies that $c(v_1, w_1)$ holds. Assume now that $c(v_i, w_i)$ holds for some $1 \leq i \leq n$. For all $v \in \{v_{i+1}, \ldots, v_n\}$, because a, b are non-zero natural numbers, we have

$$av \ > \ av_i \ = \ bw_i + c \ \geq \ bw_{i-k} + c \ (0 \leq k < i).$$

Hence $c(v, w_{i-k})$ does not hold. Similarly, for all $w \in \{w_{i+1}, \ldots, w_m\}$, $c(v_{i-k}, w)$ does not hold $(0 \leq k < i)$. The constraint $c(x, y)$ is thus still arc-consistent wrt the domains $\{v_{i+1}, \ldots, v_n\}$ and $\{w_{i+1}, \ldots, w_m\}$. Using Properties (1) and (2) again, we get that $c(v_{i+1}, w_{i+1})$ holds.

Assume now that $n < m$. Since $c(v_n, w_n)$ holds, $c(v, w_{n+1})$ does not hold for $v \in D_x$. The constraint $c(x, y)$ is thus not arc-consistent which is a contradiction. Hence $n = m$. □

The satisfiability of a system of constraints in solved form can be tested in a straightforward way.

Theorem 12 Let $S = \langle AC, DC \rangle$ be a system of constraints in solved form. S is satisfiable iff $\langle \emptyset, DC \rangle$ is satisfiable.

Proof It is clear that $\langle \emptyset, DC \rangle$ is not satisfiable iff the domain of some variable is empty in DC. If the domain of some variable is empty in DC, then S is not satisfiable. Otherwise, it is possible to construct a solution to S. By properties (1), (2), and (3), all binary constraints of S hold if we assign to each variable the smallest value in its domain. Moreover, because of node-consistency, the unary constraints also hold for such an assignment. □

It is worth noting that, in a system of constraints in solved form, all the values within the domain of a variable do not necessarily belong to a solution. For instance, in the following system

$$\langle \{x \leq y, 3x \geq 2y + 1\}, \{x \in \{2, 4, 6\}, y \in \{2, 3, 6\}\} \rangle$$

there is no solution with the value 4 assigned to x although the system is in solved form.

It remains to show how to transform a system of constraints into an equivalent one in solved form. This is precisely the purpose of the node- and arc- consistency algorithms [9].

Algorithm 13 To transform the system of constraints S into a system in solved form S':

1. apply a node-consistency algorithm to the unary constraints of $S = \langle AC, DC \rangle$ to obtain $\langle AC, DC' \rangle$;

2. apply an arc-consistency algorithm to the binary constraints of $\langle AC, DC' \rangle$ to obtain $S' = \langle AC, DC'' \rangle$.

Theorem 14 Let S be a system of constraints. Algorithm 1 produces a system of constraints in solved form equivalent to S.

We now give a complete constraint-solver for the basic constraints. Given a system of constraints S, Algorithm 15 returns *true* if S is satisfiable and $false$ otherwise.

Algorithm 15 To check the satisfiability of a system of constraints S:

1. apply Algorithm 13 to S to obtain $S' = \langle AC, DC \rangle$;
2. if the domain of some variable is empty in DC', return $false$; otherwise return *true*.

The complexity of Algorithms 13 and 15 is the complexity of arc-consistency algorithms. In [10], an arc-consistency algorithm is proposed whose complexity is $O(cd^2)$ where c is the number of binary constraints and d is the size of the largest domain. Given the form of the basic constraints, it is possible to design a specific arc-consistency algorithm whose complexity is $O(cd)$ [4] showing that basic constraints can be solved efficiently.

3.3 Reduced Form of the Basic Constraints

This subsection defines the reduced form of basic constraints in CLP(F).

Definition 16 Let σ be a consistent conjunction of basic constraints. The reduced form of σ, denoted $red(\sigma)$, is obtained as follows:

1. Let S be the system of constraints associated with σ and $S' = \langle AC', DC' \rangle$ be its solved form.
2. Let DC'' be equivalent to DC' be with only one domain constraint per variable.
3. Let AC'' be AC' without
   - the unary constraints;
   - the binary constraints $c(x, y)$ satisfying

     $$\forall v \in D_x \; \forall w \in D_y \; c(x/v, y/w)$$

     where D_x, D_y are the domains of x, y in S'.
4. $red(\sigma)$ is the conjunction of basic constraints in $\langle AC'', DC'' \rangle$.

The reduced form for the basic constraints is equivalent to the solved form since the unary constraints are node consistent (and hence implied by the domain constraints) and the binary constraints satisfying the given condition are also implied by the domain constraints.

3.4 Expressive Power

The basic constraints of CLP(F) have been chosen carefully in order to avoid an NP-Complete constraint-solving problem. For instance, allowing disequations with two variables leads to an NP-Complete problem (graph-coloring could then be expressed in a straightforward manner). Allowing equations and inequalities with three variables also leads to NP-complete problems. Finally, it should be noted that Algorithm 15 is not powerful enough to decide systems of constraints including constraints of the form $ax + by = c$.

The reason is that arc-consistency algorithms handle constraints locally while more global reasoning is necessary to check satisfiability of this class of constraints. As an example, the system

$$\langle \{x_1 + x_2 = 1, x_2 + x_3 = 1, x_3 + x_1 = 1\} \ , \ \{x_1 \in \{0,1\}, x_2 \in \{0,1\}, x_3 \in \{0,1\}\} \rangle$$

is not satisfiable although it is arc-consistent. To verify the unsatisfiability, just add the three constraints to obtain

$$2x_1 + 2x_2 + 2x_3 = 3.$$

The left member is even while the right member is odd.

4 Non-Basic CLP

The purpose of this section is to provide a systematic way to describe the operational semantics of non-basic constraints that are approximated in terms of basic constraints. As mentioned previously, approximation is present in several CLP languages.

To capture that behaviour, we introduce two new combinators in the *Ask & Tell* framework, relaxed tell and relaxed ask. These combinators are parametrized by a relaxation and an approximation function such that we are in fact defining a family of combinators.

Before starting the more formal presentation, let us give an informal account to the approach. Let σ be the accumulated constraints and assume that we face a relaxed tell on a non-basic constraint c. Relaxed tell, instead of checking the consistency of $\sigma \wedge c$ which might be quite complex in general, only checks the consistency of a relaxation of $\sigma \wedge c$. Clearly if the relaxed problem is not satisfiable, the initial problem is not satisfiable either. Moreover the solutions of the relaxed problem cannot necessarily be expressed as a conjunction of basic constraints. Hence only an approximation of the solutions, in the form of a conjunction of basic constraints, can be added to the accumulated constraints. When the approximation is not equivalent to the initial problem, the non-basic constraint cannot be removed from the goal part of the configuration. Finally, if we face a relaxed ask, then entailment is not checked wrt σ, but rather wrt to its relaxation. As a consequence, relaxed ask could return undecided (i.e. flounder) while an ask (if implemented) would have returned *true* or *false*.

4.1 Relaxation and Approximation

Relaxed tell and relaxed ask require to associate, with each non-basic constraint, (1) a relaxation function; (2) an approximation function and (3) a complete constraint-solver.

Definition 17 A *relaxation* function is a total function $r : CCR \rightarrow CCR$ such that

$$\mathcal{D} \models (\forall)(\sigma \Rightarrow r(\sigma)).$$

In other words, a relaxation function associates with each conjunction σ of basic constraints another conjunction of basic constraints that is implied by σ. Hence the relaxation of σ captures the solutions of σ and possibly some non-solutions.

We also would like to infer new basic constraints from a non-basic constraint. This is achieved through an approximation function.

Definition 18 Given a relaxation function r, an *approximation* function is a total function ap, which, given $\sigma \in CCR$ and a non-basic constraint c, returns a conjunction of basic constraints, such that $\mathcal{D} \models (\forall)((r(\sigma) \wedge c) \Rightarrow ap(\sigma, c))$.

This definition specifies that the approximation captures all solutions of $r(\sigma) \wedge c$ and possibly some non-solutions. In the following, we assume that a relaxation function and an approximation function have been defined for each constraint and denote them by the (overloaded) symbols r and ap respectively.

The non-basic constraint-solver has to satisfy a number of requirements.

Definition 19 A non-basic constraint-solver is *complete* iff it can decide

1. the consistency of c and $r(\sigma)$;
2. the entailment of c' by $r(\sigma)$

where c is a non-basic constraint, c' is a non-basic ask-constraint, and σ is a conjunction of basic constraints.

The following properties are straightforward but help understanding the transition rules.

Property 20 [Properties of relaxation and approximation]

1. $\mathcal{D} \models (\forall)((\sigma \wedge c) \Rightarrow (\sigma \wedge ap(\sigma, c)))$;
2. $\mathcal{D} \models \neg(\exists)(c \wedge r(\sigma))$ implies $\mathcal{D} \models \neg(\exists)(c \wedge \sigma)$;
3. $\mathcal{D} \models \neg(\exists)(ap(\sigma, c) \wedge \sigma)$ implies $\mathcal{D} \models \neg(\exists)(c \wedge \sigma)$;
4. $\mathcal{D} \models (\forall)(ap(\sigma, c) \Rightarrow c \wedge r(\sigma))$ implies $\mathcal{D} \models (\forall)((\sigma \wedge c) \Leftrightarrow (\sigma \wedge ap(\sigma, c)))$;
5. $\mathcal{D} \models (\exists)(ap(\sigma, c) \wedge \sigma)$ and $\mathcal{D} \models (\forall)(ap(\sigma, c) \Rightarrow c \wedge r(\sigma))$ implies $\mathcal{D} \models (\exists)(c \wedge \sigma)$;
6. $\mathcal{D} \models (\forall)(r(\sigma) \Rightarrow c)$ implies $\mathcal{D} \models (\forall)(\sigma \Rightarrow c)$.

4.2 Relaxed Tell

We are now in position to define the operational semantics of relaxed tell.

Termination: Termination of relaxed tell occurs when the approximation $ap(\sigma, c)$ is equivalent to $r(\sigma) \wedge c$ which means that $r(\sigma) \wedge c$ can be represented as a conjunction of basic constraints. Moreover if $ap(\sigma, c)$ is consistent with σ, we obtain a terminal configuration.

$$\frac{\mathcal{D} \models (\exists)(\sigma \wedge ap(\sigma, c)) \quad \mathcal{D} \models (\forall)(ap(\sigma, c) \Rightarrow (r(\sigma) \wedge c))}{\langle relax\text{-}tell(c), \sigma \rangle \longmapsto red(\sigma \wedge ap(\sigma, c))}$$

Property 20.4 and 20.5 ensure respectively the equivalence of $\sigma \wedge ap(\sigma, c)$ and $\sigma \wedge c$ and the consistency of c and σ.

Pruning: Property 20.1 allows for the addition of new basic constraints (and hence pruning of the search space) provided that $r(\sigma) \wedge c$ and $ap(\sigma, c) \wedge \sigma$ be consistent, and $ap(\sigma, c)$ be not entailed by σ.

$$\frac{\mathcal{D} \models (\exists)(r(\sigma) \wedge c) \quad \mathcal{D} \models (\exists)(\sigma \wedge ap(\sigma, c)) \quad \mathcal{D} \models \neg(\forall)(ap(\sigma, c) \Rightarrow (r(\sigma) \wedge c)) \quad \mathcal{D} \models \neg(\forall)(\sigma \Rightarrow ap(\sigma, c))}{\langle relax\text{-}tell(c), \sigma \rangle \longmapsto \langle relax\text{-}tell(c), red(\sigma \wedge ap(\sigma, c)) \rangle}$$

The third condition (non-termination) imposes to keep $relax\text{-}tell(c)$ in the resulting configuration while the last condition (non-redundancy) avoids the infinite application of the rule. Progress is achieved here together with the conjunction rules because the search space is pruned by the new constraints.

Floundering: Floundering occurs when there is no termination and when the approximation is entailed by the accumulated constraints.

$$\frac{\mathcal{D} \models (\exists)(r(\sigma) \wedge c) \quad \mathcal{D} \models \neg(\forall)(ap(\sigma, c) \Rightarrow (r(\sigma) \wedge c)) \quad \mathcal{D} \models (\forall)(\sigma \Rightarrow ap(\sigma, c))}{\langle relax\text{-}tell(c), \sigma \rangle \longmapsto flounder}$$

4.3 Relaxed Ask

Relaxed ask can be defined in a straightforward way by checking entailment of the non-basic ask-constraint by the relaxation of the accumulated constraints.

$$\frac{\mathcal{D} \models (\forall)(r(\sigma) \Rightarrow c)}{\langle \textit{relax-ask}(c) \rightarrow G, \sigma \rangle \longmapsto \langle G, \sigma \rangle}$$

$$\frac{\mathcal{D} \models (\forall)(r(\sigma) \Rightarrow \neg c)}{\langle \textit{relax-ask}(c) \rightarrow G, \sigma \rangle \longmapsto \sigma}$$

$$\frac{\mathcal{D} \models \neg(\forall)(r(\sigma) \Rightarrow c) \quad \mathcal{D} \models \neg(\forall)(r(\sigma) \Rightarrow \neg c)}{\langle \textit{relax-ask}(c) \rightarrow G, \sigma \rangle \longmapsto \textit{flounder}}$$

4.4 Relations with Ask and Tell

The soundness of the transition system can be proven by structural induction on the configurations. More interesting perhaps is the relationships between ask and tell and their relaxed versions in the case where *ask* and *tell* can be decided in the computation domain. Assume that P^* is the program P where all occurrences of relaxed ask and relaxed tell have been replaced by ask and tell. We state the following property without proof.

Property 21

- $SS(P) \subseteq SS(P^*)$
- $FS(P) \subseteq FS(P^*)$

5 Non-Basic CLP(F)

In Section 2, we have presented the basic constraints of CLP(F). These constraints can be handled by an efficient complete constraint-solver but are certainly not expressive enough to be the only constraints available in CLP(F). Moreover we have shown that apparently simple extensions to the basic constraints can lead to an NP-Complete constraint-solving problem. To complete the definition of CLP(F), it remains to specify the non-basic constraints and to define (1) the relaxations, (2) the approximations, and (3) the constraint-solvers.

Definition 22 Let σ be a conjunction of basic constraints in reduced form and $\langle AC, DC \rangle$ its associated system of constraints. The relaxation $r(\sigma)$ is the conjunction

$$x_1 \in D_1 \wedge \ldots \wedge x_n \in D_n$$

such that $DC = \{x_1 \in D_1, \ldots, x_n \in D_n\}$.

In other words, the relaxation in CLP(F) simply ignores all constraints but the domain constraints. Note also that computing the relaxation does not induce any cost since the accumulated constraints are already in reduced form for incrementality purpose.

There are two approximation functions depending if full k-consistency is achieved (i.e. ap_1) or if the reasoning is only performed on the minimum and maximum values in the domains (i.e. ap_2). For some symbolic constraints, both approximations might be useful depending upon the problem at hand.

Definition 23 Let σ be a conjunction of basic constraints and c be a non-basic constraint with variables $x_1, \ldots, x_n$ such that $\mathcal{D} \models (\exists)(r(\sigma) \wedge c)$. Let $dom(r(\sigma), c)$ be the set of natural number tuples

$$\{\langle a_1, \ldots, a_n \rangle \mid \mathcal{D} \models (\exists)((r(\sigma) \wedge c)[x_1/a_1, \ldots, x_n/a_n]\}).$$

The approximation $ap_1(\sigma, c)$ is defined by

$$x_1 \in D_1 \wedge \ldots \wedge x_n \in D_n,$$

and the approximation $ap_2(\sigma, c)$ is defined by

$$x_1 \in \{min_1, \ldots, max_1\} \wedge \ldots \wedge x_n \in \{min_n, \ldots, max_n\},$$

where D_i represents the projection of $dom(r(\sigma), c)$ along its argument i, and min_i and max_i represents the minimum and maximum value in D_i.

Finally, the constraint-solvers for the non-basic constraints are once again based on consistency techniques. However they use the semantics of the constraints to come with an efficient implementation. For instance, inequalities and equations use a reasoning about variation intervals [5, 16]. There can be a large variety of (numeric and symbolic) constraints that might be considered as interesting primitive constraints so that we will not specify the constraint-solvers for them. Note only that the constraint-solvers can be made complete as only domain constraints (i.e relaxation of basic constraints) are considered in conjunction with a non-basic constraint. Also, in [17], we propose a new combinator that makes possible to define most interesting numerical and symbolic constraints from a small set of primitive constraints.

6 Conclusion

This paper has presented an extension to the *Ask & Tell* framework to capture, in a simple and precise way, the operational semantics of CLP languages where some constraints are approximated, inducing an incomplete constraint-solver. The extension consists of two new combinators, relaxed tell and relaxed ask, that are parametrized on a relaxation and approximation function. Constraint are divided into two sets, basic and non-basic constraints. Basic constraints (that can be decided efficiently) are handled as usual while non-basic constraints (that are approximated in terms of basic constraints) are handled through the new combinators.

The extended framework has been instantiated to CLP over finite domains. The basic constraints of CLP(F) have been identified and arc-consistency has been shown to be the basis of an efficient and complete constraint-solver. The relaxation and approximation functions for non-basic constraints in CLP(F) have also been described.

The contributions of this paper include (1) a precise and simple definition of the operational semantics of CLP(F) and (2) the definition of two new combinators that should allow for a precise operational semantics of several other CLP languages.

A structural operational semantics does not describe how to implement the language. What remains to be described for that purpose is how the constraint-solvers are implemented and under which conditions a non-basic constraint is reconsidered after floundering. An efficient arc-consistency algorithm for basic constraints is defined in [4]. The wakening of non-basic constraints is based on a generalization of the techniques used for delay mechanisms (e.g. [11, 2]). Both issues are beyond the scope of this paper.

Future work includes the study of the properties of the above operational semantics as well as its relationships with the declarative and denotational semantics. Application of the approach to other CLP languages with an incomplete constraint-solver will also be considered.

Acknowledgments

Vijay Saraswat gave the initial motivation beyond this paper. His comments were instrumental in showing the value of the SOS semantics. Discussions with Baudouin Le Charlier and the CHIP team of ECRC are also gratefully acknowledged.

References

[1] W Buttner and al. A General Framework for Constraint Handling in Prolog. Technical report, Siemens Technical Report, 1988.

[2] A. Colmerauer. PROLOG II: Manuel de Reference et Modele Theorique. Technical report, GIA - Faculte de Sciences de Luminy, March 1982.

[3] A. Colmerauer. A Generalized Implicit Enumeration Algorithm for Graph Coloring. *CACM*, 28(4):412–418, 1990.

[4] Y. Deville and P. Van Hentenryck. Arc-Consistency for a Class of Networks. Technical report, CS Department, Brown University, 1990. (Forthcoming).

[5] M. Dincbas, H. Simonis, and P. Van Hentenryck. Extending Equation Solving and Constraint Handling in Logic Programming. In MCC, editor, *Colloquium on Resolution of Equations in Algebraic Structures (CREAS)*, Texas, May 1987.

[6] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japon, December 1988.

[7] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, FRG, January 1987.

[8] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[9] A.K. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.

[10] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[11] L. Naish. *Negation and Control in Prolog.* PhD thesis, University of Melbourne, Australia, 1985.

[12] W Older and A. Vellino. Extending Prolog with Constraint Arithmetics on Real Intervals. In *Canadian Conference on Computer & Electrical Engineering*, Ottawa, 1990.

[13] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, CS Department, University of Aarhus, 1981.

[14] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1989.

[15] P. Van Hentenryck. A Framework for Consistency Techniques in Logic Programming. In *IJCAI-87*, Milan, Italy, August 1987.

[16] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[17] P. Van Hentenryck and Y. Deville. A new Logical Connective and its Application to Constraint Logic Programming. Technical report, CS Department, Brown University, 1990. (Forthcoming).

[18] P. Voda. The Constraint Language Trilogy: Semantics and Computations. Technical report, Complete Logic Systems, North Vancouver, BC, Canada, 1988.

[19] C. Walinsky. CLP($\sum^*$). In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.