UNIVERSITE D'ORLEANS

Faculté des Sciences

LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

The Constraint System of the Grammatical View of (Constraint) Logic Programming

Gérard Ferrand, Arnaud Lallouet, Julien Arsouze
Université d'Orléans, LIFO

Rapport N^o **2002-06**

The Constraint System of the Grammatical View of (Constraint) Logic Programming

Gérard Ferrand, Arnaud Lallouet, Julien Arsouze

Université d'Orléans — LIFO
BP 6759
F-45067 Orléans cedex 2
`Firstname.Name@lifo.univ-orleans.fr`

Abstract :
Constraint Logic Programming (CLP) consists in programming with relations. Some are predefined and are called constraints and some are defined by the user's logic program. Depending on which one we focus on, CLP can be seen from two points of view. Our goal in this paper is to propose a theoretical model based on the ideas of the Grammatical View of Logic Programming [3]. It presents a uniform view of a CLP computation both at the program and constraint levels. The essence of our framework is to materialize a solution proof as the solution of tree constraints and to link them with the program constraints. This makes possible a tight integration of finite domain constraints and CLP inside a unified framework of solver collaboration.

keywords: CLP; CSP; chaotic iterations; (C)SLD resolution

1 Introduction

Constraint Logic Programming (CLP) consists in programming with relations. Some are predefined and are called constraints and some are defined by the user's logic program. Depending on which one we focus on, CLP can be seen from two points of view.

In the first point of view, called the *CLP scheme* [6, 12, 7], a CLP(D) program is a logic program enhanced with constraints on a domain D. The computation is done by a syntactic accumulation of constraints in a store guided by the building of a proof called *derivation* by (C)SLD resolution. The solver is here a black-box solver which rejects stores and is subject to incompleteness: an inconsistent store may remain unrejected.

In the second point of view, or *domain reduction scheme*, originating from finite domain Constraint Satisfaction Problems (CSPs) [15, 14], Logic Programming is just a host language, a way to add new constraints in the store. The central part of the computation is the solver's task which consists in reducing the domains associated to the constraints until it hopefully finds a solution or an inconsistency. An incompleteness appears when the solver is not able to compute the exact domains but only an approximation. However, in finite domains, complete labeling on all variables is able to ensure completeness.

Both visions yield a two levels formalism (the one of primitive constraints and the one of user-defined constraints) with a minimal interaction surface which has several drawbacks. It makes the computation thread harder to understand or to follow by automatic methods like debuggers. This is illustrated by the difficulty to develop debugging tools for Constraint Programming. For example, in the Discipl ESPRIT Project [2], two kinds of debugging were considered with little interaction. Also, it prevents or limits for the compiler the possibility of using information from a level to optimize the other.

However, for logic programs, i.e. CLP(Herbrand) with equality constraints, the Grammatical View of Logic Programming [3] already aims at taking this central place between the two levels. One of the main idea is that the accumulation of constraints in the store is structured by a tree made out of the program's clauses. This tree is called *skeleton* (of a proof tree) because it holds the minimal quantity of information. From the clause found at each of its node can be reconstituted the proof tree up to a renaming. Then the search for an answer for a goal can be viewed as the collaboration of two separate tasks: the building of the skeleton and the solving of its associated store[1].

Our goal in this paper is to go a step further in this direction by considering the task of building the skeleton as constraint solving too. This is done in three steps: we propose a constraint system for the skeleton part, whose solutions are well-formed skeletons, we link this constraint system to the store and we apply generic constraint solving methods to the resulting system. The structuration of the program remains the same at the user's level but the execution model is changed since it allows the two levels to be freely interleaved. In the original Grammatical View of Logic Programming, the construction of a solution tree has been only detailed for top-down computations and the two levels are clearly distinguished. Our framework can be viewed as a particular solver collaboration. In particular, this makes possible a tight integration of finite domain constraints and CLP inside a single framework. We illustrate this point by sketching the specialization of our framework for a Finite Domain solver.

In section 2, we recall some definitions about CSPs and CLP, in particular concerning the chaotic iteration framework [5, 1], some extensions to cope with non-finite number of variables and search [9] and the Grammatical View of Logic Programming [3]. In section 3, we detail the constraint system of the skeleton part of a solution to a program and its links with the usual constraint store and the integration of paradigms is discussed.

2 Preliminaries

The aim of this section is to set the picture by recalling the underlying framework we use in the remaining of the paper: CSPs [15, 14] and the operational semantics

[1] Note that the terminology of Grammatical View of Logic Programming comes from attributed grammars because of the analogy between programs and grammars and between skeletons and derivation trees.

of their resolution [15, 5, 1, 9] on one side, CLP [6] and the Grammatical View of Logic Programming [3] for the other.

2.1 CSPs

Let $A = (A_i)_{i \in I}$ be a family of sets. We denote $\Pi A = \Pi_{i \in I} A_i = \{(a_i)_{i \in I} \mid \forall i \in I, a_i \in A_i\}$ the *indexed* cartesian product of the family A. If all A_i are ordered sets, ΠA is ordered by the product ordering of all A_i. For $J \subseteq I$ and a family a on I, we denote by $a|_J = (a_i)_{i \in J}$ the restriction of a to the set of indices J. Moreover, we denote by A^J the indexed cartesian product $\Pi(A|_J)$. With this notation, we can write ΠA as A^I and the notation coincides with the usual cartesian power $I \longrightarrow B = B^I$ when all A_i are equal to the same B. This allows to use multiple domains transparently. We denote by $\mathcal{P}(A)$ the powerset of A.

Let V be a denumerable set of variables and $D = (D_x)_{x \in V}$ be a family of domains. A *constraint* is a couple $c = (W, T)$ where $W = var(c) \subseteq V$ is the *arity* of c and $T = sol(c) \subseteq D^{var(c)}$ is a table containing the tuples solution of c. A *constraint satisfaction problem*, or in short *CSP*, is a set C of constraints. In the classical A.I. literature, CSPs are most of the time supposed finite, so we rather use the name *constraint system*, or CS, to refer to possibly infinite sets of constraints. In this paper, the finiteness of constraint systems is not assumed. The particular case of finite domain constraint solving classically involves a finite domain D. A *solution* is a tuple of D^V satisfying all constraints, i.e. $s \in sol(C) \iff \forall c \in C, s|_{var(c)} \in sol(c)$. Solving a CS means finding one (or all) of its solutions. It could be done only by enumeration but the combination of search and approximation methods called *consistencies* yields far better performances in practice.

CSP solving by approximation is mainly a matter of reduction of the extension of the constraints [15, 13, 5, 1]. It is useful to consider that every variable x belong to an unary constraint d_x, also called its domain by a slight abuse of language. Variable domains are most of the time the only reduced constraints, as in the CHIP [15] or the Gnu-Prolog system [4]. Thus a *search state* is defined by a set of values for each variable which represents the current reduction state of the possible assignments for the variable. The *search space* of the CSP C is therefore $\mathcal{SS} = \Pi_{x \in V} \mathcal{P}(D_x)$. It is ordered by the product ordering of set inclusion for each domain.

A *consistency* is simply a property of search states. The best known consistency is the so-called *arc-consistency* [11] which holds for a binary constraint, say $c(X, Y)$, if, for each value d in the domain of X, there is a corresponding value e in the domain of Y such that $c(d, e)$ holds (and conversely by exchanging the role of X and Y). This notion is extended straighforwardly to hyperarc-consistency for n-ary constraints. A whole CSP is arc-consistent if its variable domains are arc-consistent for all constraints. A solution is found when every variable domain has been reduced to a singleton and we call such a state *singletonic*. Conversely, if a variable domain becomes empty, then the CSP has no solution.

All known consistencies can be represented by the common closure $C_{\mathcal{F}}(s)$ of a set $\mathcal{F}$ of *reduction* operators, i.e. monotonic, correct and contracting operators

[13, 5, 1]. Let $s = (s_x)_{x \in V}$ be a search state. For the arc-consistency with respect to $c(X, Y)$, it can be formalized by the mapping $r(s) = \{d \in s_x \mid \exists e \in s_y, c(d, e)\}$ and the same one obtained by exchanging the role of x and y. A rule r obviously defines an associated reduction operator. The set of operators on $\mathcal{SS}$ extending these mappings completely defines the expected consistency, which is obtained by computing their greatest common fixed-point. This can be done by defining an aggregate operator $F(s) = \bigcap_{f \in \mathcal{F}} f(s)$ and iterating it [5], or by a *chaotic iteration* [5, 1], where only one operator of $\mathcal{F}$ is applied at each time, fairness of the scheduling ensuring the confluence. The *standard iteration* of $\mathcal{F}$ from $s \in \mathcal{SS}$ is the sequence $a = (a_i)_{i \in \mathbb{N}}$ of elements of $\mathcal{SS}$ where $a_0 = s$ and $\forall i > 0$, $a_i = F(a_{i-1})$.

Definition 1 (Chaotic iteration).
A chaotic iteration *of $\mathcal{F}$ starting from $s \in \mathcal{SS}$ is a sequence $b = (b_i)_{i \in \mathbb{N}}$ of elements of $\mathcal{SS}$ where $b_0 = s$ and $\forall i \in \mathbb{N}^+$, $\exists f \in \mathcal{F}, b_i = f(b_{i-1})$.*

A chaotic iteration is *complete* if its limit is stable. In such a case, its limit is the closure $C_{\mathcal{F}}(s)$. Chaotic iterations may differ depending on the strategy used to schedule the operators. A chaotic iteration is *fair* if no operator is delayed for ever: $\forall i \in \mathbb{N}, \exists j \in \mathbb{N}$, such that $\forall f \in \mathcal{F}, e_j \subseteq f(e_i)$. The main property of chaotic iterations is their confluence:

Theorem 1 (Chaotic iteration, Apt [1]).
Let $a = (a_i)_{i \in \mathbb{N}}$ be the standard iteration of $F_{\mathcal{F}}$ and $b = (b_i)_{i \in \mathbb{N}}$ be a fair chaotic iteration, each of them starting from x. Then $\lim(b) = \lim(a)$.

This confluence result means that two fair chaotic iterations have the same limit, which is moreover the same as the standard iteration.

2.2 Chaotic derivations

In [9], we proposed two extensions of this framework: γ-chaotic iterations to cope with the halting of chaotic iterations and chaotic derivations to model the interleaving with the search mechanism.

In chaotic iterations, any strategy is acceptable. However it can be taken advantage of certain properties of the reduction functions in order to accelerate the convergence. For example, functions used to enforce arc-consistency are usually idempotent, so once applied, it is unnecessary to consider them immediately after. This is why existing algorithms and systems implement a data structure to schedule the reduction functions: it may be a queue as in AC-3 [16], a priority queue as in Gnu-Prolog [4] or a more complex event parsing structure as in the *Choco* system [8] in which cheap propagations are done immediately while costly ones are further delayed.

We propose to model theses classes of strategies as a *choice restriction set* $\gamma \subseteq \mathcal{P}(\mathcal{F})$ which associates to a computation state a subset γ of "choosable" reduction functions:

Definition 2 (choice restriction set).
A set $\gamma \subseteq \mathcal{P}(\mathcal{F})$ *is a* choice restriction set *for a search point* $s \in \mathcal{SS}$ *if* $\forall f \in \mathcal{F} - \gamma, f(s) = s$.

In other words, any operator outside of γ does not reduce the domains. Note that a search point s for which $\gamma = \emptyset$ is stable by $\mathcal{F}$. This can be used operationnally to detect stability. At each search point s, there exists a minimal set $\gamma(s) = \{f \in \mathcal{F} \mid f(s) \neq s\}$. But conversely, a computed γ may include non-reducing operators.

A γ-chaotic iteration is a kind of chaotic iteration for which, at each step i, the reduction function is chosen in γ_i. Let I be a set of indices (i.e. $\mathbb{N}$ or $[1..n]$) and $I^+ = I - \{0\}$.

Definition 3 (γ-chaotic iteration).
A γ-chaotic iteration *of* $\mathcal{F}$ *starting from* $s \in \mathcal{SS}$ *is a sequence* $c = (c_i, \gamma_i)_{i \in I}$ *with* $c_i \in \mathcal{SS}$ *and* γ_i *is a choice restriction set where* $c_0 = s$ *and* $\forall i \in I^+$, $\exists f \in \gamma_{i-1}$ *such that* $c_i = f(c_{i-1})$.

The set γ has two advantages: it models the propagation queue used in some consistency algorithms and $\gamma = \emptyset$ is a sufficient condition for an iteration to be complete. The same confluence result as for chaotic iterations holds for γ-chaotic iterations with a modified definition of fairness [9].

In order to really find a solution, steps of local consistency (called here *reduction*) and searching are interleaved. The efficiency of constraint programming resides in the good balance by which these two efforts are combined. Usually constraint programming systems propagate until reaching the fixpoint to ensure consistency after each search step. We propose here a finer modelization in which the search mechanism appears at the operator's level. It yields that more interleaving strategies can be described.

In order to find solutions, we need to focus on subparts of the closure; to ensure completeness, we need to explore them in a systematic manner. Only standard labeling is described in this paper. In other words, a search step corresponds to the tentative of assignment of a variable with all possible values in its current domain. This leads to the following notion of search tree. There is two kinds of nodes: a reduction node corresponds to the application of a reduction operator and a labeling node corresponds to the decision of exploration of the subsequent search space by labeling:

Definition 4 (Search tree node).
A search tree node is a triple (d, α, β) *where:*

- d *is the current search point*
- α *determines the nature of the next step and is either:*
  - *a choice restriction set* γ *if the node is a reduction node. This kind of node has one son which corresponds to the application of an operator in* γ.
  - *a variable name* $x \in VN$ *if the node is a labeling node. Such a node has as many sons as there are elements in the current domain of* x.

– *β precises the exact path from the father to the current node and is either:*
  - *the function f which was applied to get to the current node if the father was a reduction node.*
  - *the element $e \in d_x$ which was chosen in this branch if the father was a labeling node.*
  - *a dummy element $\Box$ for the root.*

A *search tree* is a tree made out of search tree nodes and meeting the above constraints. Its branches may be finite or infinite.

Example 1. The following tree illustrates a reduction step and then a labeling step:

$$\begin{array}{ccc} & (d^0, \gamma_0, \Box) & \\ & | & \\ & (d^1, x, f) & \\ \swarrow & \downarrow & \searrow \\ (d^2, \gamma_2, e_1) & (d'^2, \gamma_2, e_2) & (d''^2, \gamma_2, e_3) \end{array}$$

In this example, the tree is such that $f \in \gamma_0$, $d^1 = f(d^0)$ and $\forall i \in [1..3], e_i \in d^1_x$. The exact domain obtained after a labeling step will be precised further.

A *search step* consists in a decrease in the search space obtained by taking the intersection of the current element $s \in \mathcal{SS}$ and some arbitrary element $z = (z_x)_{x \in VN} \in \mathcal{SS}$ called a *search filter*. The application of this search filter reduces the domain s_x to $s_x \cap z_x$. We denote by $s \cap z$ the search point $(s_x \cap z_x)_{x \in VN}$ (pointwise intersection).

For our special case of labeling, the search filter consists to assign a value to a variable while leaving the other domains unchanged. For a variable $x \in VN$ and a value $e \in D_x$, let define the *search filter* $\mathrm{lf}(x, e) = (l)_{y \in VN}$ by

– $l_x = \{e\}$.
– $\forall y \neq x \in VN, l_y = D_y$.

With this definition, the domain part of a search tree node which is child of a labeling node (d, x, β) can be defined as $(d \cap \mathrm{lf}(x, e), \alpha, e)$. We can make the rest of the theory with a more general notion of search filter $z \in SS$.

The following definition is motivated by the formalization of a branch of a search tree, which is a sequence of reduction and search steps. A chaotic derivation is a sequence of application of operators and search filters:

Definition 5 (Chaotic derivation).
Let Z be the set of search filters. A chaotic derivation *of $(\mathcal{F}, Z)$ starting from $x \in \mathcal{SS}$ is a sequence $d = (d_i)_{i \in I}$ of elements of $\mathcal{SS}$ where:*

– $d_0 = x$.
– $\forall i \in I^+$, *either*
  - $\exists f \in \mathcal{F}$ *such that* $d_i = f(d_{i-1})$, *or*

- $\exists z \in Z$ *such that* $d_i = d_{i-1} \cap z$.

When considering a branch of ST, each labeling step can be viewed as the application of a particular operator $\mathrm{lo}(x,e) : \mathcal{SS} \longrightarrow \mathcal{SS}$ defined by $lo(x,e)(s) = s \cap lf(x,e)$. This operator is monotonic and contractant but is in general incorrect: the focus is made on one part of the search space. We call it a *labeling operator*. For a branch b of ST, we denote by LO the set of its labeling operators. A chaotic derivation is *complete* if its limit is stable by $\mathcal{F} \cup LO$. Note that a chaotic iteration is a kind of chaotic derivation for which no search filter is used. Conversely, a chaotic derivation is a kind of chaotic iteration for the set of operators $\mathcal{F} \cup LO$.

Completeness of the search is achieved by ensuring the complete exploration of the search space, for example by considering for a variable labeling all the possible values which remain in its domain.

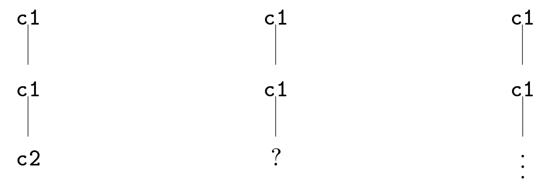
2.3 CLP

CLP(D) consists in replacing in Logic Programming unification by constraint solving on a domain D (pure LP is viewed as CLP(*Herbrand*) with the constraint $=$). This feature consists in defining a language to express the constraints and its integration into Logic Programming [6]. The constraint language is classically defined on a set of variables V and a signature containing a set Σ of function symbols and a set Π_C of constraint predicate symbols. A primitive constraint is an atomic formula $c(t_1, \ldots t_n)$ where $c \in \Pi_C$ and the t_i's are (Σ, V)-terms. A constraint is a particular case of first-order (Π_C, Σ, V)-formula. The pre-interpretation of the constraints is given by a (Π_C, Σ)-structure $\mathcal{D}$ on the domain D.

Let Π_P be a set of program predicates symbols. Let $arity(p)$ denote the arity of p. An atom is an atomic (Π_P, Σ, V)-formula. For an atom $a = p(t_1, \ldots, t_n)$, let $pred(a)$ denote the symbol predicate of a. The interpretation of a predicate p is given by a set of pseudo-atom called D-atom $p(d_1, \ldots, d_n)$ where the d_i's are elements of D. The D-base is the set of all D-atoms and we denote it by $\mathcal{B}$. A $\mathcal{D}$-interpretation is identified to an arbitrary subset of the D-base. A valuation is a mapping from V to D.

A definite clause is a formula $h \leftarrow c, b_1, \ldots, b_n$ where h and the b_i's are atoms and c is a constraint. We call n the *arity* of the clause. A $\mathcal{D}$-instance of such a clause is a rule obtained by instanciation by a valuation satisfying the constraint c. A definite program is a set of definite clauses and is identified to the set of $\mathcal{D}$-instances of its clauses for the algebraic semantics and its associated fixed-point semantics. To a given program P is associated an immediate consequence operator $T_P : \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B})$ defined by $T_P(I) = \{h \mid \exists h \leftarrow B \in P, B \subseteq I\}$. This operator is monotonic and continuous and its least fixed-point (in short lfp) is the least $\mathcal{D}$-model of P [6]. It is convenient to assume that clauses are given in a normal form in which all atoms only contain different variables and all connexion between variables are postponed in the constraint part. For example, the clause `f(N,N*M) :- N>0, f(N-1,M)` is written `f(A,B) :- A>0, B=A*D,` C=A-1, f(C,D). For a clause cl, let $head(cl)$ denote its head atom, $arity(cl)$ denote its arity, $constr(cl)$ denote its constraint part and $body(cl, i)$ denote the i-th atom in its body.

The notion of constraint domain [6] adds to $\mathcal{D}$ a theory $\mathcal{T}$ and a solver $\mathcal{S}$ which should agree in representing D. Basically, a solver is a decision procedure that answers `true` or `false` to a given satisfaction problem. An interesting aspect of the solver is its possible incompleteness: for certain satisfaction problems, the solver cannot decide whether it is satisfiable or not and answers `unknown`. The most famous example is the real solver of the system *CLP*($\mathcal{R}$) [7] which solves linear constraints and delays non-linear ones until they hopefully become linear according to the instanciation of variables. The solver enjoys some properties such as independence with respect to renaming and constraint ordering, and monotonicity (well-behaved solver of [6]). For simplicity purposes, we only model the solver as a *rejection criterion* rc, which is a set of (rejected) stores such that $false \in rc$.



(a) Complete skeleton (b) Partial skeleton (c) Complete infinite skeleton

Fig. 1. The "skeletal" view of CLP

In [6], the operational semantics is expressed in terms of derivation, which can also be viewed as the building of a proof tree. This proof tree can be abstracted by keeping at its nodes the only useful information, namely the clause used to build the node. This object is called *skeleton* in [3]. A skeleton associated to a maximal success derivation is *complete*. A skeleton associated to a non-terminal derivation state is a *partial skeleton*. It still holds leaf places to be filled with clause names. These places are denoted by "?". Consider the following program:

```
c1: p(X) :- p(f(X)).
c2: p(f(a)).
```

Three examples of skeletons built using these clauses are presented in figure 1. A proof tree can be reconstituted from a skeleton by choosing a renaming for all nodes such that no two clauses share a common variables. Such a renaming is precisely defined relatively to an occurence in the skeleton. Then all clauses of the skeleton glued together define its associated store.

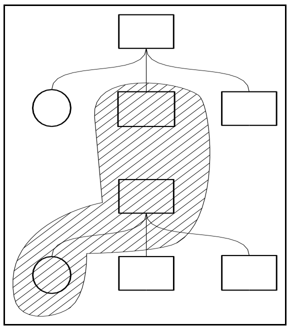


Fig. 2. Constraint grouping in a skeleton (circle=constraint, box=atom)

An occurence is a finite word on $\mathbb{N}_+$. Let $OCC = (\mathbb{N}_+)^*$ denote the set of occurences, ϵ the empty word and . the concatenation. A *tree domain* is a set $TD \subseteq OCC$ of occurences which satisfies the properties of trees. Let $\mu \in OCC$ be an occurence and $i \in \mathbb{N}_+$:

- $\epsilon \in TD$.
- $\mu.i \in TD \Longrightarrow \mu \in TD$.
- $(\mu.i \in TD$ and $1 \leq j \leq i) \Longrightarrow \mu.j \in TD$.

The occurence μ is the *father* of $\mu.i$ and an occurence μ such that $\forall i \in \mathbb{N}_+, \mu.i \notin TD$ is called a *leaf* of TD. Let $\mathcal{L}$ be a set of *labels* and let $\mathcal{L}_\perp = \mathcal{L} \cup \{\perp\}$ where $\perp \notin \mathcal{L}$. A *tree* $t = (t_\mu)_{\mu \in TD}$ is a family of labels indexed by the tree domain TD (or equivalently a function from TD into $\mathcal{L}$ mapping an occurence μ to a label $t(\mu)$). A tree $t : TD \rightarrow \mathcal{L}$ can be classically extended to a function $OCC \rightarrow \mathcal{L}_\perp$ such that $\forall \mu \notin TD, t(\mu) = \perp$. For a tree t, $dom(t)$ denotes its domain ($dom(t) = \{\mu \in OCC \mid t(\mu) \neq \perp\}$) and $leaves(t)$ denotes the set of its leaves.

A skeleton is a mapping $sk : OCC \rightarrow P_\perp$. If we assume that for each variable name x appearing in P and for each occurence $\mu \in OCC$, there exits a variable x^μ in V, let ψ_μ be the renaming $x \mapsto x^\mu$. We define:

- $constr(sk, \mu) = \psi_\mu(constr(sk(\mu)))$.
- $head(sk, \mu) = \psi_\mu(head(sk(\mu)))$.
- $body(sk, \mu, i) = \psi_\mu(body(sk(\mu), i))$.
- $equal(sk, \mu, i)$ denotes the constraint $body(sk, \mu, i) = head(sk, \mu.i)$. This constraint is illustrated by the two boxes in figure 2.

The store associated to a skeleton sk is then:

$$store(sk) = constr(sk, \epsilon) \wedge \bigwedge_{\mu.i \in dom(sk)} constr(sk, \mu.i) \wedge equal(sk, \mu, i)$$

Moreover, the derivation up to a given state can be reconstituted from the skeleton, up to a renaming, with an additional information: the strategy or *choice rule* used to choose the atom.

3 CLP's CSP

The execution of a program aims at finding a complete success skeleton. In this part, we propose to model declaratively this notion as the solution of a certain constraint system.

3.1 A constraint system for proof skeletons

We propose to model inside a unique constraint system both the proof skeleton and the constraints of the program. Since this constraint system is rather complex, we decompose our approach in steps:

1. for each occurence ν, we define a variable X_ν whose domain is the set of clauses $P_\perp$. We call such a variable a *variable occurence*. A assignment of all variable occurences gives a complete skeleton.
2. variable occurences are submitted to tree constraints (TREE):
   (a) root: $X_\epsilon \neq \perp$
   (b) brother: $X_{\mu.i} = \perp \rightarrow X_{\mu.i+1} = \perp$
   (c) child: $X_\mu = \perp \rightarrow X_{\mu.1} = \perp$
3. the connexion between clauses is made by equalities between atoms of a node and of its children. But although in a skeleton the clauses are all available to describe the connexion, it is not the same in the constraint system in which at each occurence all clauses are considered as equally possible. Hence we need a more systematic way of naming the store variables to be able to set equalities without knowing which clause is assigned to an occurence. From now on, we assume that the programs' clauses are of the form:

   $$h(X_{0,1}, \ldots, X_{0,n_0}) \text{ :- } c, b_1(X_{1,1}, \ldots, X_{1,n_1}), \ldots, b_m(X_{m,1}, \ldots, X_{m,n_m})$$

   For each occurence $\mu \in OCC$, let V^μ be the set of variables $\{X^\mu_{i,j} \mid i \in \mathbb{N}, j \in \mathbb{N}^+\}$ and let ρ_μ be the renaming such that $\rho_\mu(X_{i,j}) = X^\mu_{i,j}$. As for occurence variables, the domain of a store variable has to be extended by $\perp$ if the variable is actually not used.
4. store variables are submitted to list constraints: once a variable $X^\mu_{i,j}$ is not used, so are $X^\mu_{i,k}$ with $k > j$. Let LIST(μ, i) be the constraint:

   $$\text{LIST}(\mu, i) \quad \equiv \quad \bigwedge_{j \in \mathbb{N}^+} \left(X^\mu_{i,j} = \perp \Rightarrow X^\mu_{i,j+1} = \perp\right)$$

5. choosing a clause cl at an occurence μ has four consequences:
   - the variable occurence X_μ is assigned to cl,
   - the children occurences up to the clause's arity are different of $\perp$,
   - the next variable occurence is assigned to $\perp$,
   - the store variables beyond the arity of each atom are unused, either for the head or for the atoms in the body of the clause. This is taken into account by setting the first unused variable to $\perp$ and using the LIST constraint.

Hence, let $\text{CLAUSE}(\mu, cl)$ be the constraint:

$$\begin{aligned}\text{CLAUSE}(\mu, cl) \quad \equiv \quad & X_\mu = cl \\ & \wedge\ \rho_\mu(constr(cl)) \\ & \wedge \bigwedge_{i \in [1..arity(cl)]} X_{\mu.i} \neq \perp \\ & \wedge\ X_{\mu.arity(cl)+1} = \perp \\ & \wedge \bigwedge_{i \in [1..arity(cl)+1]} X^{\mu}_{i,arity(body(cl,i))+1} = \perp \\ & \wedge\ \text{LIST}(\mu, i)\end{aligned}$$

6. an occurence can be labeled by any clause or $\perp$. This is expressed by the following constraint:

$$\text{CHOICE}(\mu) \quad \equiv \quad (X_\mu = \perp) \vee \bigvee_{cl \in P} \text{CLAUSE}(\mu, cl)$$

7. when a clause cl has been chosen for an occurence $\mu.i$, the connexion with its father is needed. Let $\text{CONNECT}(\mu.i, cl)$ be the constraint:

$$\begin{aligned}\text{CONNECT}(\mu.i, cl) \quad \equiv \quad & pred(head(X_{\mu.i})) = pred(body(X_\mu, i)) \\ & \wedge \bigwedge_{j \in [1..arity(head(cl))]} X^{\mu.i}_{0,j} = X^{\mu}_{i,j}\end{aligned}$$

   Note that the static naming of the store variables is needed. This constraint is illustrated by the dashed lines in figure 2.
8. at each occurence except ϵ, the constraint is built using CHOICE and CONNECT:

$$\text{ASSIGN}(\mu) \quad \equiv \quad \text{CHOICE}(\mu) \wedge \text{CONNECT}(\mu)$$

9. the root occurence ϵ is like the others except that there is no connexion with the father:

$$\text{ROOT} \quad \equiv \quad \text{CHOICE}(\epsilon)$$

   Note that the TREE constraints will prevent X_ϵ to be equal to $\perp$.
10. finally the whole constraint system can be given using these constraints:

$$\text{ALL} \quad \equiv \quad \text{TREE} \wedge \text{ROOT} \wedge \bigwedge_{\mu.i \in OCC} \text{ASSIGN}(\mu.i)$$

11. A solution to the constraint ALL determines a skeleton and a solution of its associated store and thus an atom in the least $\mathcal{D}$-model of P. This atom is root of a proof tree having this skeleton.

In this constraint system, a solution expresses at a single level a successful skeleton and its non-rejected store. Since solutions are supported by skeletons, the set of constraints can also be described by the (much shorter) formula:

$$\bigvee_{sk \text{ finite skeleton}} \left[\left(\bigwedge_{\nu \in dom(sk)} X_\nu = sk(\nu) \right) \wedge store(sk) \right]$$

But because of the disjunction, the practical resolution of such a constraint system would be a kind of "generate and test" on skeletons, untractable in practice. An amazing remark is that this is the use of the data structure of tree occurences which leads to extend the domain from finite to infinite tree. This latter and preferred constraint system cannot be represented in the constraint system ALL because there is no way to express the finiteness of trees.

3.2 An approximation scheme

The system ALL can be solved by approximation. Indeed, occurence variables are finite domain variables (since P is finite). Here follows a set of reduction rules which can be used to narrow the domain of each variable. These rules are given in a rather informal way: their lhs is seen as a condition for the rule to apply, and their rhs as an action. Hence for example, $X_\mu = \perp \Rightarrow X_{\mu.i} = \perp$ means that if the domain of $X_\mu = \perp$, then the domain of $X_{\mu.i}$ is included in $\{\perp\}$. Operationnally, an intersection is made between the current domain of $X_{\mu.i}$ and $\{\perp\}$, possibly leading to an inconsistency (the domain of $X_{\mu.i}$ becomes empty). The lhs is a guard which prevents an operator to be used too early and is a way to ensure the monotonicity of the operators.

1. TREE:
   - root: $X_\epsilon \neq \perp$
   - brother: $\forall \mu \in OCC, X_{\mu.i} = \perp \rightarrow X_{\mu.i+1} = \perp$
   - child: $\forall \mu \in OCC, X_\mu = \perp \rightarrow X_{\mu.1} = \perp$
2. LIST: $\forall \mu \in OCC, \forall i \in \mathbb{N}, \forall j \in \mathbb{N}^+, X^\mu_{i,j} = \perp \Rightarrow X^\mu_{i,j+1} = \perp$
3. CLAUSE: when a clause is chosen, three actions can be performed. The lhs is each time the same and corresponds to the case when the domain of X_μ has been reduced to a singleton: $\forall X_\mu \in OCC, X_\mu = \{cl\} \Rightarrow$.
   - $\forall i \in [1..arity(cl)], X_{\mu.i} \neq \perp$, or, to be more precise, $X_{\mu.i}$ is included in the definition clauses of $pred(body(X_\mu, i))$
   - $X_{\mu.arity(cl)+1} = \perp$
   - $\forall i \in [1..arity(cl)+1], X^\mu_{i,arity(body(cl,i))+1} = \perp$
4. DOMAIN: other operators on store variables have to be defined and depend on the domain solver. This is the support for various specialization of our work.
5. LABELING: labeling operators are used to try a clause at a certain occurence. They are used to explore systematically the disjunction present in CHOICE constraints. This exploration defines a search tree (which coincides with the SLD search tree for the solving of the constraint system with the CSLD strategy exposed below).

As for CSPs, the operational semantics of this constraint system is the scheduling of operators by chaotic iteration. At each step an operator is chosen and executed in random order. When the closure has been reached, a complete skeleton is built on the occurences. Note that due to the chaotic execution of the operators, the skeleton can be built by assigning a clause to some internal node before its father or even before the root. If it corresponds to a valid skeleton, then this validity will be ensured when disconnected parts will eventually connect. This feature was already present in an informal way in the Grammatical View of Logic Programming [3] where skeletons can be assembled by pieces from smaller skeletons floating in space.

3.3 CSLD as a CS solving strategy

In this context, CSLD can be expressed by a particular operator scheduling strategy. It is convenient to introduce here the set γ of "choosable" operators [9] which simply models in this case the resolvant, i.e. the atoms which are waiting to be chosen. The following associations are meaningful:

- (C)SLD resolution may be seen as an ad-hoc strategy which follows the form of the CS. In particular, the backtracking-based search of (C)SLD comes from the simple treatment of the CHOICE disjunction. Intelligent backtracking methods are different strategies, as well as the use of delays. A topic deserving further attention would be the import of constraint specific heuristics in this context, For example, the use of the first-fail principle amounts to choose the atom with the least remaining possible clauses to develop the search tree.
- The iteration is complete when $\gamma = \emptyset$, or in other terms, when the resolvant is empty. Then the CSLD resolution ends with a success, the reduced search point is singletonic and contains a finite complete skeleton.
- Fairness of the chaotic iteration is exactly fairness of the CSLD resolution. This is of particular importance for infinite derivations which yield infinite complete skeletons.
- Infinite derivations build infinite non-rejected skeletons. A fair infinite (C)SLD derivation is an infinite fair chaotic derivation and the reduced domain is also singletonic and contains a infinite complete skeleton. From the confluence theorem for chaotic derivations $[5, 1, 9]$ follows the independence of non-failure branches. As a consequence, the result of independence of success answers in Logic Programming [10] and CLP [6] can be retrieved.

3.4 Specialization for CLP(FD)

Specialization for FD consists in defining more precisely the DOMAIN operators. For a black-box solver, the operators defining the rejection criterion are somehow artificial and deal with finite parts of the store. On the contrary in the FD case, rejection of a store occurs when the domain reductions lead to an empty domain for some variable. It can be an occurence variable signifying that the skeleton construction is wrong or a FD variable submitted to classical reduction operators. All operators are scheduled at the same level and can be freely interleaved.

4 Conclusion

In this paper, we presented an original model which aims at bridging the gap between the program and constraint part in CLP(D) programs. This is an unifying framework which allows to clarify the main advantage of the Grammatical View of Logic Programming, namely the clear separation between the construction of the skeleton of the proof tree and the possible satisfaction of its associated store of constraints.

References

1. K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
2. Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski, editors. *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project).* Number 1870 in LNCS. Springer, 2000.
3. Pierre Deransart and Jan Maluszyński. *Grammatical view of Logic Programming.* MIT Press, 1993.
4. Daniel Diaz and Philippe Codognet. The GNU Prolog system and its implementation. In *ACM Symposium on Applied Computing*, volume 2, pages 728–732, Villa Olmo, Como, Italy, 2000.
5. François Fages, Julian Fowler, and Thierry Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37(1-3):185–212, 1998.
6. Joxan Jaffar, Michael Maher, Kim Marriott, and Peter Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.
7. Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The $CLP(\mathcal{R})$ language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.
8. François Laburthe and the OCRE project. Choco: implementing a CP kernel. In *TRICS, Techniques foR Implementing Constraint programming Systems, CP 2000 post-conference workshop*, Technical report TRA9/00, Singapore, sep 2000.
9. Arnaud Lallouet, Gérard Ferrand, and Julien Arsouze. Chaotic derivations : an operational semantics for complete constraint solving. In Slim Abdennadher and Thom Frühwirth, editors, *RCORP'01, post ICLP/CP'01 workshop on Rule-Based Constraint Reasoning and Programming*, Paphos, Cyprus, 2001.
10. John W. Lloyd. *Foundations of logic programming.* Springer-Verlag, second edition, 1987.
11. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
12. Kim Marriott and Peter Stuckey. *Programming with Constraints: an introduction.* MIT Press, 1998.
13. Ugo Montanari and Francesca Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.
14. Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
15. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
16. D.L. Waltz. *Understanding line drawings in scenes with shadows*, pages 19–91. McGraw Hill, 1975.