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/

Chaotic iterations for constraint propagation and labeling

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

Rapport N^o **2000-4**

Chaotic iterations for constraint propagation and labeling

Julien Arsouze, Gérard Ferrand, Arnaud Lallouet
Université d'Orléans — LIFO
BP 6759 — 45067 Orléans cedex 2
{Firstname.Name}@lifo.univ-orleans.fr
http ://www.univ-orleans.fr/SCIENCES/LIFO/Members/{*name*}

March 23, 2000

Abstract

Here is an attempt to define a uniform semantic framework for constraint solving based on chaotic iterations and downward closure operators, taking into account both propagation and labeling. We extend the formalism of Montanari and Rossi [10] with more general rules of two kinds: while relaxation rules narrow the search space and preserve solutions, new restriction rules split the search space to explore it piece by piece. We show that independence of the computed approximation extends in some sense to labeling. Moreover, this framework can be used to model various approaches including the glass-box one used for finite domain constraint solving in Gnu-Prolog.

1 Introduction

Constraint programming is a high-level formalism used to declaratively model both the problem and the strategy to solve it. Constraint solving aims at finding consistent values for a set of variables linked together by a set of constraints. This operation commonly involves the traversal of large search spaces with blind or informed algorithms. One of the major issue of constraint programming is the notion of filtering, a preprocessing technique that drastically reduces the size of the search space without changing its solutions, allowing blind search algorithms to be actually used in real-sized applications.

These algorithmic techniques are usually named constraint propagation and share the common goal of wishing to eliminate values that do not participate to any solution. They are interleaved with a traversal of the search space, usually named labeling, in order to find a solution. The central idea of *local consistency* is to compute an approximation of the solutions which is correct (no solution are omitted) and smaller than the original search space. On the other side, labeling cuts out a CSP into smaller pieces which are easier to traverse. It is well-known that labeling alone leads to the inefficient "generate and test" algorithm while propagation alone only returns in general an imprecise approximation.

Because of the inherent complexity of these mechanisms, such programs are very difficult to analyze, verify or debug. This paper is an attempt to define a uniform semantics framework for constraint solving, with these problems in mind.

Under different forms, the notion of *chaotic iteration* has been recognized to be an unifying framework to model constraint propagation, and the notion of *closure* operator to characterize the result in a computation-independent way. Montanari and Rossi [10] relate local consistencies to the common greatest fixpoint of a set of operators. They also noticed the relevance of many algorithms used to enforce a consistency to a general chaotic iteration algorithm, although they did not mention this name. The same kind of framework has been studied by Codognet and Diaz [6], Benhamou [5], Van Emden [15] and Apt [2] among others.

The main contribution of this paper is a co-inductive semantics which takes into account both propagation and labeling. Co-induction comes from the use of downward closure operators. We first model, following [10], the different consistencies by a set of rules R, being R-consistent meaning being *stable* by the rules. Our rules are general enough to model not only propagation, but also labeling. Hence we distinguish two kinds of rules: relaxation rules that reduce the search space while maintaining the set of solutions, and restriction rules that cut the search space into pieces in order to explore it more easily. Besides the classical result of independence of the computed approximation by chaotic iteration, we also give an additional independence result in the presence of labeling.

The obtained framework generalizes Montanari and Rossi's one and allows to model the glass-box approach [6] (originating from [16]) and used as a basis for the finite domain constraint solving of Gnu-Prolog [7].

This paper is organized as follows: section 2 summarizes basic principles on closures and chaotic iterations, emphasizing the use of co-induction. Section 3 introduces reduction rules which model both relaxation and labeling, and states the main independence result. Section 4 instanciates the general reduction rules to solution-preserving relaxation rules for solving CSPs and defines the computed approximation. Section 5 is devoted to comparisons with related works.

2 Closures and chaotic iteration

In this section, we recall in an abstract framework some useful results of fixpoint theory: properties of closures, iteration of operators and fast computation of the common fixpoint of a set of operators by chaotic iteration. We denote by $\mathcal{P}(A)$

the powerset of A.

Closures. Let $(E, \subseteq)$ be a complete lattice. An operator t on E is contractant if $t(x) \subseteq x$. Let R be a finite set of *monotonic* and *contractant* operators on E. Some of the following results can be proved in a more general setting than complete lattices (see [2] for instance) but we do not need this feature since our purpose is to apply them to a product of set orderings. Moreover, we need the existence of both upper and lower bounds to define the closure.

Definition 2.1 (Stable element)
An element $x \in E$ is stable *by R if $\forall t \in R$, $x = t(x)$.*

We also say that x is R-consistent.

Lemma 2.2 (Closure)
For all $x \in E$, there exists $C_R(x)$ the closure *of x by R, which is the greatest element $y \subseteq x$ of E stable by R. Thus we have:*

- $C_R(x) \subseteq x$
- $\forall t \in R, C_R(x) = t(C_R(x))$
- $(y \subseteq x$ *and* $\forall t \in R, y = t(y)) \Longrightarrow y \subseteq C_R(x)$

Proof Take $C_R(x) = \bigcup\{y \mid y \subseteq x$ and y stable by $R\}$. ⊛

In fixpoint theory, the system $y \subseteq x$ and $\forall t \in R, y = t(y)$ is called a *co-inductive* definition of $C_R(x)$ (see [9, 12] for example). Usually, t is supposed to be only monotonic but we can get a contractant operator from a monotonic one by considering $t(y) \cap y$. The principle of *proof by co-induction* is the following: let $y \subseteq x$; in order to prove that $y \subseteq C_R(x)$, it is sufficient to prove $\forall t \in R, y \subseteq t(y)$. The correction of this method follows from the properties of $C_R(x)$. Moreover, if $y \subseteq C_R(x)$, its completeness is ensured by the existence of a $z \subseteq x$ such that $y \subseteq z$ and z stable by R (at least $z = C_R(x)$!).

Lemma 2.3 (Closure operator)
C_R is a downward closure operator, that is $C_R(x)$ is monotonic, contractant and idempotent. Moreover, x stable by R is equivalent to $x = C_R(x)$.

Proof Straightforward. ⊛

The following lemmas follow from the characteristics of C_R:

Lemma 2.4

1. $C_{R_1 \cup R_2}(x) = C_{R_1 \cup \{C_{R_2}\}}(x) = C_{\{C_{R_1}, C_{R_2}\}}$ *("compositionality" lemma).*
2. $R \subseteq R' \Longrightarrow C_{R'}(x) \subseteq C_R(x)$.
3. $C_{R_1 \cup R_2}(x) \subseteq C_{R_1}(C_{R_2}(x))$.
4. *if* $C_{R_1}(C_{R_2}(x))$ *is stable by* R_2*, then* $C_{R_1}(C_{R_2}(x)) = C_{R_1 \cup R_2}(x)$.

Proof Most of them use the proof method by co-induction.

1. Let $A = C_{R_1 \cup R_2}(x)$ and $B = C_{R_1 \cup \{C_{R_2}\}}(x)$. We first prove that $A \subseteq B$ by co-induction. By definition, A is stable by all operators in $R_1 \cup R_2$. It is sufficient to have $\forall t \in R_1 \cup \{C_{R_2}\}, A \subseteq t(A)$, which is already true for $t \in R_1$. So let $t = C_{R_2}$, to have $A \subseteq C_{R_2}(A)$, it is sufficient to have $\forall t' \in R_2, A \subseteq t'(A)$, which is also true.

   In the other side, it is sufficient to have $\forall t \in R_1 \cup R_2, B \subseteq t(B)$. B is stable by R_1 and by C_{R_2}, hence $B = C_{R_2}(B)$. So it means that B is stable by all operators in R_2.

   To get the second equality, just apply the first one to $R_2 = \{C_{R_1}\}$.

2. Let $A = C_R(x)$ and $B = C_{R'}(x)$. In order to get $B \subseteq A$, it is sufficient to have $\forall t \in R, t(B) \subseteq B$. But $\forall t \in R', t(B) = B$ and $R \subseteq R'$. Hence $B \subseteq A$.

3. $R_2 \subseteq R_1 \cup R_2$, hence, by lemma 2.4-2, we have $C_{R_1 \cup R_2}(x) \subseteq C_{R_2}(x)$. We know that C_{R_1} is monotonic, hence $C_{R_1}(C_{R_1 \cup R_2}(x)) \subseteq C_{R_1}(C_{R_2}(x))$. But $C_{R_1 \cup R_2}(x)$ is stable by R_1, hence is equal to $C_{R_1}(C_{R_1 \cup R_2}(x))$.

4. $C_{R_1}(C_{R_2}(x))$ is stable by R_2 by hypothesis and obviously stable by R_1. Hence $C_{R_1}(C_{R_2}(x)) \subseteq C_{R_1 \cup R_2}(x)$.

$\circledast$

Chaotic iteration. In order to compute the closure, we can combine all operators of R into one and iterate this operator until stability. Let T_R be the operator defined by $T_R(x) = \bigcap_{t \in R} t(x)$. Clearly, T_R is still monotonic and contractant and $x = T_R(x)$ is equivalent to x is stable by R.

An iteration of T_R from x is the (possibly) infinite sequence $x^0, x^1, \ldots, x^i, \ldots$ of elements of E where $x^0 = x$ and $\forall i > 0$, $x^i = T_R(x^{i-1})$. We write x^ω the element $\bigcap_{i \geq 0} x^i$. If $\subseteq$ is a well-founded ordering, the sequence stabilizes for some k such that $x^k = x^{k+1} = x^\omega = C_R(x)$.

From now on we only consider such orderings.

When the computation of T_R is found too expensive, a technique called *chaotic iteration* can be used. It consists in applying each time only one $t \in R$ instead of T_R. The definition of the iteration is the same except that $\forall i > 0$, $x^i = t_i(x^{i-1})$ where $t_i \in R$. Due to the contractance property of the operators, we have $x^0 \supseteq x^1 \supseteq \ldots \supseteq x^i \supseteq \ldots \supseteq x^k = x^{k+1} = x^\omega$. We call s the sequence $t_1, \ldots, t_i, \ldots$ of chosen operators. The following theorems state the correctness and completeness of the method with regards to the computation of the closure of x by R.

Theorem 2.5 (Correctness)
If, for some $k \geq 0$, x^k is stable by R, then $C_R(x) = x^k$.

Proof

- $\subseteq$: by recurrence, $C_R(x) \subseteq x^0$ and for $i > 0$, if $C_R(x) \subseteq x^{i-1}$, $C_R(x) = t_i(C_R(x)) \subseteq t_i(x^{i-1}) = x^i$.

- $\supseteq$: because $C_R(x)$ is the greatest set stable by R.

⊛

The existence of such a k is not guaranteed. To enforce it, we need a stronger hypothesis on the way the operators are chosen during the iteration. As a matter of fact, we need to be sure that no operator has been forgotten for the stability check. This hypothesis is known as *fairness* and imposes that each operator appears infinitely often in the sequence.

Theorem 2.6 (Completeness)
If the sequence s is fair, that is $\forall t \in R, \forall k \geq 0, \exists j > k$ such that $t = t_j$, then x^ω is stable.

Proof We want x^ω to be stable by R, that is $\forall t \in R$, $x^\omega \subseteq t(x^\omega)$. Let us take some operator t in R. For some $k \geq 0$, $x^\omega = x^k$. Then, there exists a $j > k$ such that $t = t_j$. $j - 1 \geq k$, hence $x^{j-1} \subseteq x^k$. Thus $x^\omega \subseteq x^j = t_j(x^{j-1}) \subseteq t_j(x^k) = t(x^\omega)$. ⊛
A short historical perspective of this technique is given in [2].

3 Reduction operators and chaotic derivations

The process of solving a CSP consists informally in narrowing its search space until it hopefully reaches the solutions. But the general problem is too complex to be directly attacked. Hence several techniques known as *consistencies* have been proposed to give an approximation of the solutions. But in order to really obtain a solution, this mechanism has to be supported by a cutting out of the domains that takes the hand when relaxation fails to give a solution.

In the following, we mainly use the notion of family and *indexed* cartesian product: let A be a set and I be a finite set of indices, we denote by $(a_i)_{i \in I}$ where $a_i \in A$ a family of elements of A indexed by I. Let $A = (A_i)_{i \in I}$ be a family of sets. We denote $\Pi(A) = \Pi_{i \in I} A_i = \{a = (a_i)_{i \in I} \mid \forall i \in I, a_i \in A_i\}$ the *indexed* cartesian product of the family A. If all A_i are ordered sets, $\Pi(A)$ is ordered by the product ordering of all A_i. For $J \subseteq I$ and a family a on I, we denote by $a|_J$ 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 $\Pi(A)$ 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 take advantage of the generality of multiple domains without having to explicitly deal with its complications.

This section introduces abstract objects we call *constraints*. A constraint is defined by its name and a set of elements (that will be tuples) we call *extension* of the constraint. At this stage, only names and extensions of the constraints are useful to define the search space. In particular, variables are not precised. For example, an abstract formalism such as the "X in r" (or glass-box) scheme [16, 6] considers expressions on variables domains which are unary constraints. As a matter of fact, variables are only here to refer to their domain.

Let IC be a set of indices called *constraint names* and $\mathcal{D} = (\mathcal{D}_c)_{c \in IC}$ be a family of sets. We call *search space* the set $\mathcal{SS} = \Pi_{c \in IC}(\mathcal{P}(\mathcal{D}_c))$. Thus for an element $x \in \mathcal{SS}$, we have $x = (x_c)_{c \in IC}$ where $x_c \subseteq \mathcal{D}_c$. We still note $\subseteq$ the product ordering on $\mathcal{SS}$. When applied to "real" constraints, IC is intended to define the constraints of the CSP and x to be a family of sets of tuples defining the constraints.

Definition 3.1 (Reduction rule)
Let $S \subseteq IC$ and $a \in S$ be a distinguished constraint. A function $r : \Pi_{c \in S}(\mathcal{P}(\mathcal{D}_c)) \longrightarrow \mathcal{P}(\mathcal{D}_a)$ is a reduction rule *of type (S, a) if:*

- *r is monotonic*
- *$\forall x_S \in \Pi_{c \in S}(\mathcal{P}(\mathcal{D}_c))$, we have $r(x_S) \subseteq x_{S_a}$*

An important remark is that we do not impose for a rule to be correct since there is still no notion of solution. A rule r is extended on IC to get an operator called $reduc_r$:

Definition 3.2 (Reduction operator)
For $x \in \mathcal{SS}$, the element $reduc_r(x) = x'$ is defined as follows:

- *$x'_c = x_c$ for $c \neq a$*
- *$x'_a = r(x|_S)$*

Lemma 3.3
The operator $reduc_r$ is monotonic and contractant.

Proof Straightforward. ⊛
All monotonic and contractant operators we consider in this framework are instances of $reduc_r$ for some rule r. So, if R is a set of rules, we denote by C_R the operator $C_{\{reduc_r \mid r \in R\}}$.

Some rules of particular nature are distinguished now in order to model labeling. They express some cutting out of the extension of a constraint and we call them *restriction rules*:

Definition 3.4 (Restriction rule)
Let $a \in IC$ and $Z \subseteq \mathcal{D}_a$. The restriction rule *defined by (a, Z) is the rule of type $(\{a\}, a)$ such that $r(x_a) = x_a \cap Z$.*

We denote its associated reduction operator by $restrict_{(a,Z)}$. By definition $restrict_{(a,Z)}$ is idempotent, so it is a downward closure. We now consider a set of these rules. For each $a \in IC$, let $\mathcal{P}_a \subseteq \mathcal{P}(\mathcal{D}_a)$ be a set of subsets of $\mathcal{D}_a$. As in set theory, $\bigcap \mathcal{P}_a$ denotes the intersection of the elements of $\mathcal{P}_a$ (by convention, if $\mathcal{P}_a = \emptyset$, then $\bigcap \mathcal{P}_a = \mathcal{D}_a$).

The next two lemmas focus on restriction rules. They state that the only restriction that does matter is the intersection of all sets involved in the cutting out.

Lemma 3.5
For each $a \in IC$, we have $C_{\{restrict_{(a,Z)} \mid Z \in \mathcal{P}_a\}} = C_{\{restrict_{(a,\bigcap \mathcal{P}_a)}\}}$.

Proof We have to prove that $C_{\{restrict_{(a,Z)} \mid Z \in \mathcal{P}_a\}}(x) = C_{\{restrict_{(a,\bigcap \mathcal{P}_a)}\}}(x)$. This is true because $(\forall Z \in \mathcal{P}_a, x_a = x_a \cap Z) \iff x_a = x_a \cap \bigcap \mathcal{P}_a$. ⊛
This result is valid for a cutting out of all constraints:

Lemma 3.6
$C_{\{restrict_{(a,Z)} \mid a \in IC, Z \in \mathcal{P}_a\}} = C_{\{restrict_{(a,\bigcap \mathcal{P}_a)} \mid a \in IC\}}$.

Proof By lemma 2.4-1 and lemma 3.5 because $\{restrict_{(a,Z)} \mid a \in IC, Z \in \mathcal{P}_a\} = \bigcup_{a \in IC} \{restrict_{(a,Z)} \mid Z \in \mathcal{P}_a\}$. ⊛
Now we add to our restriction operators a set R of monotonic and contractant operators (when composed of operators correct with respect to the set of solutions, this set is intended to model relaxation). The result still holds:

Lemma 3.7
$C_{R \cup \{restrict_{(a,Z)} \mid Z \in \mathcal{P}_a\}} = C_{R \cup \{restrict_{(a,\bigcap \mathcal{P}_a)}\}}$.

Proof Again by lemma 2.4-1 and lemma 3.5. ⊛
The simultaneous existence of reduction and restriction operators allows to represent the complex interaction between relaxation and labeling steps that occur in the process of constraint solving. At each step, a choice is made between a possible reduction rule or a restriction. The important point is that a restriction is not determined statically. The only thing that matters is the final intersection of all restriction that have occured. We call such a sequence of elements of the search space a *chaotic derivation*:

Definition 3.8 (Chaotic derivation)
Let R be a fixed set of reduction rules. A chaotic derivation *cd from x with regards to R is a sequence $x^0, x^1, \ldots, x^i, \ldots$ such that $x^0 = x$ and for $i > 0$, either:*

- $x^i = reduc_{r_i}(x^{i-1})$ *for some* $r_i \in R$

- $x^i = restrict_{(a_i,Z_i)}(x^{i-1})$ *for some* $a_i \in IC$ *and* $Z_i \subseteq \mathcal{D}_{a_i}$

A chaotic derivation may be success or failure:

Definition 3.9 (failure and success derivation)
A chaotic derivation $x^0, x^1, \ldots, x^i, \ldots$ *is a:*

- *failure if for some* $k \geq 0$*, we have* $x^k_{a_k} = \emptyset$.
- *success if for some* $k \geq 0$*,* x^k *is stable by* R *and not a failure.*

In each case, a derivation is finite and terminates at x_k. Note that we do not require all extensions of constraints to be empty to detect a failure, nor stability. This condition is nevertheless enough to model the absence of solution of the CSP when the extension of a constraint is empty.

The following theorem expresses the independence of the resulting approximation with respect to the effective cutting out of the extensions and gives a co-inductive meaning to the computed solutions of a CSP:

Theorem 3.10
Let $\mathcal{P}_a$ *be exactly the set of all* $Z \subseteq \mathcal{D}_a$ *that appear in the success chaotic derivation cd terminated by* x^k*. We have:*

$$x^k = C_{R \cup \{restrict_{(a,Z)} \ \mid \ a \in IC, Z \in \mathcal{P}_a\}}(x) = C_R(C_{\{restrict_{(a, \bigcap \mathcal{P}_a)} \ \mid \ a \in IC\}}(x))$$

Proof First we remark that x^k is stable by R and obviously stable by the restrictions. Hence x^k is obtained by a chaotic iteration which computes the closure by all these operators. This proves the first equality. The second one is obtained with the application of lemma 3.6 followed by the lemma 2.4-4, taking into account that $C_R(C_{\{restrict_{(a,\bigcap \mathcal{P}_a)} \ \mid \ a \in IC\}}(x))$ is stable by the used restrictions. ⊛

This theorem is analogous to the theorem of independence of the computation rule in (C)LP [8]. It states that a success does not depend on the order of application of the reduction rules. To be more precise, it states that the computed success x^k only depends on R and on the family $(\mathcal{F}_a)_{a \in IC}$ where each $\mathcal{F}_a = \bigcap \mathcal{P}_a$. Note that $\mathcal{F}_a$ does not depend on R. Hence, x^k does not depend on the order of application of the rules of R nor on the exact set $\mathcal{P}_a$ giving the intersection $\mathcal{F}_a = \bigcap \mathcal{P}_a$.

Note that $\mathcal{F}_a$ may be any set included in $\mathcal{D}_a$, between two "extreme" situations: $\mathcal{F}_a = \mathcal{D}_a$ if no labeling is applied and $\mathcal{F}_a$ reduced to a singleton for a complete labeling. Our notion of labeling can be located anywhere between these two extremes since we allow partial labeling to occur. Partition and covering of the set are as well two special cases.

Now, let us consider a restriction step in a chaotic derivation: a restriction rule defined by (a, Z) has to be applied. Assume we want to try all choices among

$Z = Z_1, \ldots, Z = Z_m$ where $Z_1, \ldots, Z_m \subseteq \mathcal{D}_a$. For example, classical labelings use a partition of $\mathcal{D}_a$, but we are not restricted to this case. In fact, the chosen set do not have to be disjoint, they even do not have to cover the whole sets $\mathcal{D}_a$.

If we consider such choices for all restriction steps, all our chaotic derivations are organized as a tree, which is the analogous of the SLD tree in (C)LP [8]. We call this tree a *chaotic derivation tree*. If l is a success leaf of a chaotic derivation tree, we denote by $\mathcal{F}_{a,l}$ the corresponding intersection set $\bigcap \mathcal{P}_a$ for the success derivation ending at the leaf l. We call this object a *final restricted set*.

As a consequence of the previous theorem, we can state that if two derivation trees have the same final restricted sets $\mathcal{F}_{a,l}$, then their computed successes are the same. Note that the final restricted sets do not depend on the reduction rules of R. This result is similar to the following formulation of the independence of the computation rule in (C)LP: two SLD trees have the same success answers [8]. This result is obvious for a complete labeling (where the final restricted sets are singletons) but is still valid for any partial labeling.

4 Constraint satisfaction problems and relaxation

This section relates chaotic derivations to constraint solving. Basically, there are two kinds of reduction: restrictions for labeling and other reduction which are expected to satisfy an additionnal property which is to be correct with respect to the solutions.

First, we define a CSP in the style of [10]:

Definition 4.1 (CSP)
A Constraint Satisfaction Problem $\mathcal{C}$*, or in short* CSP *is given by a 5-uple* (IV, IC, co, D, x) *composed of a syntactic part:*

- *a finite set* IV *of variable indices (or names).*
- *a finite set* IC *of constraint indices.*
- *a* connection function $co : IC \longrightarrow \mathcal{P}(IV)$ *that associates to a constraint the set of its connected variables.*

and a semantic part:

- *a family* $D = (D_i)_{i \in IV}$ *of non-empty finite domains.*
- *a family of sets of tuples* $x \in \Pi_{c \in IC} \mathcal{P}(D^{co(c)})$ *called the* extension *of the CSP. This means that* $x = (x_c)_{c \in IC}$ *associates a set of tuples on the appropriate variables domains to every constraint symbol in* IC*. We call* $x_c \subseteq D^{co(c)}$ *the* extension *of* c*.*

One can remark that the search space $\mathcal{SS} = \Pi_{c \in IC}\mathcal{P}(D^{co(c)})$ is obtained by instanciation of the abstract $\mathcal{D}_c$ by the cartesian product $D^{co(c)}$ of its variable domains. In the following, we are interested in computing (an approximation of) the solution of a given CSP (IV, IC, co, D, x). Hence we omit to mention it.

Now we define the solutions for $S \subseteq IC$, which are naturally the set of all tuple satisfying the constraints included in a given extension:

Definition 4.2 (Solution)
For $S \subseteq IC$ and $x_S \in \Pi_{c \in S}\mathcal{P}(D^{co(c)})$, let

$$Sol(x_S) = \{u \in D^{IV} \mid \forall c \in S, u|_{co(c)} \in x_{S_c}\}$$

We denote by $Sol(x_S, a) = \{u|_{co(a)} \mid u \in Sol(x_S)\}$ the projection of the solutions on some constraint $a \in S$. Note that $Sol(x_S, a) \subseteq x_{S_a}$. In particular, the global solutions of the CSP whose extension is $x \in \mathcal{SS}$ are given by $Sol(x)$. Our purpose is to establish an effective comparison between the solutions of a CSP and the actual approximation computed by C_R. So we need to define an intermediate object representing the solutions and comparable to C_R. We call it $SolvedForm$:

Definition 4.3 (Solved form)
For $x \in \mathcal{SS}$, let $SolvedForm(x) = x'$ be defined by $\forall c \in IC, x'_c = Sol(x, c)$.

Note that again $Sol(x, c) \subseteq x_c$. $SolvedForm$ is a CSP whose extension is exactly the solutions of the original CSP, as stated by the following lemma:

Lemma 4.4
$Sol(SolvedForm(x)) = Sol(x)$.

Proof Let $SolvedForm(x) = x'$.

- $\subseteq$: by monotonicity of Sol.
- $\supseteq$: let $u \in Sol(x)$. Then, $\forall c \in IC, u|_{co(c)} \in Sol(x, c) = x'_c$. Hence $u \in Sol(SolvedForm(x))$.

⊛

A relaxation rule is a reduction rule which preserves the solutions:

Definition 4.5 (Relaxation rule)
A rule r of type (S, a) is a relaxation rule if, for $x_S \in \Pi_{c \in S}\mathcal{P}(D^{co(c)})$, we have $Sol(x_S, a) \subseteq r(x_S)$.

Note that we do not require the relaxed constraint to be unary. However, this important case has led [15] to define the so-called *value constraints* sub-scheme.

According to definitions 4.5 and 3.1, for $x \in \mathcal{SS}$, we have $Sol(x|_S, a) \subseteq r(x|_S) \subseteq x_a$. In [10], the set of solutions is the very definition of the relaxation rule: they take $r(x|_S) = Sol(x|_S, a)$.

In the present paper, the associated reduction operator is also called *relaxation operator* and preserves the solutions. We denote it by $relax_r$, which is in fact some $reduc_r$ when r is a relaxation rule.

Lemma 4.6
$Sol(relax_r(x)) = Sol(x)$.

Proof Let $relax_r(x) = x'$.

- $\subseteq$: by monotonicity of Sol.
- $\supseteq$: let $u \in Sol(x)$. Then, $\forall c \in IC, u|_{co(c)} \in x_c$. Hence $\forall c \neq a$, we have $u|_{co(c)} \in x'_c$. But $u|_{co(a)} \in Sol(x|_S, a) \subseteq r(x|_S) = x'_a$.

⊛
Because relaxation rules preserves the solutions, $SolvedForm$ is stable by each relaxation operator:

Lemma 4.7
Let $relax_r$ be a relaxation operator. Then $SolvedForm(x)$ is stable by $relax_r$.

Proof Since $relax_r$ is a contractant operator, we only need to verify that $SolvedForm(x) \subseteq relax_r(SolvedForm(x))$. Let $SolvedForm(x) = x'$ and r be a rule of type (S, a). Only the extension of a is changed by $relax_r$, so we want to prove that $Sol(x, a) \subseteq r(x'|_S)$.
A relaxation is correct ($Sol(x|_S, a) \subseteq r(x|_S)$), so we need to prove that $Sol(x, a) \subseteq Sol(x'|_S, a)$, thus it is sufficient to have $Sol(x) \subseteq Sol(x'|_S)$. Let $u \in Sol(x)$, then $\forall c \in S, u|_{co(c)} \in Sol(x, c)$. But $Sol(x, c) = x'_c$. Hence $u \in Sol(x'|_S)$. ⊛

Corollary 4.8
If R is a set of relaxation rules, then $SolvedForm(x) \subseteq C_R(x)$.

Proof $C_R(x)$ is the greatest set stable by R. ⊛

Corollary 4.9
$Sol(x) = Sol(C_R(x))$.

Proof

- $\supseteq$: by contractance of C_R.
- $\subseteq$: because $Sol(x) = Sol(SolvedForm(x)) \subseteq Sol(C_R(x))$.

$\circledast$

This last result can also be justified in a computational way by considering a chaotic iteration.

Now, we can consider also restriction rules, in order to define a labeling and the results of section 3 apply. But if we want to preserve the set of solutions, we expect the chaotic derivation tree to satisfy an additional condition: at each restriction step, the cutting out of $\mathcal{D}_a$ defined by $Z_1, \ldots, Z_m$ has to be a covering of $\mathcal{D}_a$, that is $Z_1 \cup \ldots \cup Z_m = \mathcal{D}_a$.

5 Example and related work

This section presents an example and shows how our formalism can apply to get some other frameworks.

5.1 Comparison with Montanari and Rossi

First, we would highlight the importance of [10] as a foundation of this stream of work. In order to characterize certain computation that use each operator only once (the so-called "perfect relaxations"), they set up most of the basic notions in a graph-theoretical framework: closure and stability with regards to a set of operators, chaotic iterations and independence of the computed relaxation with regards to the chaotic iteration. Our generalization, besides the integration of labeling, consists in a generalisation of the relaxation operators. We do not require our operators to really compute the set of local solutions but simply to be correct with regards to them. As a consequence, we retrieve all results of [10] by taking the following correspondence:

- The *connection graph* is what we call the syntactic part of the CSP where nodes are variables and arcs are constraints.
- The *labeling function*, also called *definition function* in [11] is what we call the extension of the CSP.
- The presence of a distinguished constraint forces to identify a CSP or a sub-CSP to a rule with a predefined semantics: the computation of the solutions. In contrast, our rules are simply based on some syntactic structure of the CSP (rule of type (S, a) of definition 3.1). This allows to clearly separate the syntax (the constraints names upon which our rule ranges) from the semantics (the reduction function associated to a rule which is simply correct and not forced to be equal).

Thus, the rules of [10] written $a \longleftarrow S$ can be defined in our framework as $r(x_S) = Sol(x_S, a)$, according to definitions 3.1 and 4.2. As an example, we can define the relaxation rules for arc consistency. Assume there exists for each variable v an unary constraint d_v representing its domain. Then, for each binary constraint $c_{a,b}$, we have the rule of type $(\{d_a, d_b, c_{a,b}\}, d_a)$ whose semantics is given by $r(x_S) = Sol(x_S, d_a)$. The rest of [10] goes further into the description of specific algorithms such as arc and path consistency maintaining algorithms and perfect relaxation.

5.2 Example

Let X and Y be two variables on the domain $\{1, 2\}$ and let $X \neq Y$ be the only constraint. We choose to use the arc-consistency rules. Hence d_X and d_Y are the unary constraints which represent respectively the domain of X and Y. The set R is composed of the two following rules for which we use the same syntax and meaning as in [10]:

- $d_X \leftarrow X \neq Y,\ d_X,\ d_Y$
- $d_Y \leftarrow X \neq Y,\ d_Y,\ d_X$

Let first consider a relaxation according to the rules of R without labeling. The variables domains remain unchanged. Let now add a partial labeling PL which only enumerates the values of X. Because there was no domain reduction by using relaxation rules, PL is composed of two restriction rules : $PL_1 = (d_X, \{1\})$ and $PL_2 = (d_X, \{2\})$. This cutting out yields two CSPs in which X is instanciated. The constraint $X \neq Y$ may now be activated. We get finally the two following success leaves :

- $d_{X_1} = \{1\}$ and $d_{Y_1} = \{2\}$
- $d_{X_2} = \{2\}$ and $d_{Y_2} = \{1\}$

5.3 Comparison with the "X in r" scheme

Our generalization of the scheme of [10] allows to take into account a more general approach which has led to a state-of-the-art implementation: the so-called "X in r" or "glass-box" approach [16, 6] used in the Gnu-Prolog [7] system, among others. In this scheme, a *store* (in the sense of [6]) is a finite set of basic constraints "X in r" where X is a FD (finite domain) variable and r is an expression denoting a *range* which is a subset of $\{0, 1, \cdots, infinity\}$ (*infinity* is a particular integer denoting the greatest value a variable can take).

To express the glass-box scheme in our framework we replace each FD variable X by a *constraint name* ($\in IC$) and we define its extension by $\mathcal{D}_X = \{0, 1, \ldots, infinity\}$, $x_X = dom(X)$ representing the current domain of X. The value of a (monotonic) expression r is represented in our framework by a *reduction rule*, denoted also by r, of type (S, X). Note that S includes X and all the variables Y having an occurrence in the expression r (in an *indexical domain* *dom*(Y) or an *indexical term* *min*(Y) or *max*(Y)). It is easy to see that a computation in this scheme is a chaotic iteration using only reduction operators as defined above.

One of the most interesting aspect of this framework is that, from the basic "X in r" constraints, it is possible to define high-level user constraints like, for example:

```
'x≥y'(X,Y) :- X in min(Y)..infinity, Y in 0..max(X).
```

In the GNU-Prolog [7] system, such constraints are predefined in a library. A preprocessor is used to translate them at compile time but it is nevertheless possible for the user to access to this mechanism and to define new constraints in such a declarative way. Clearly, the semantics of such definitions is formalized by a *closure* in our framework.

But some of the rules are *not* monotonic, like "X in -dom(Y)" used to model the difference and where the symbol - denotes a complementation. In such a case, the activation of the rule (the "tell" of the constraint) is delayed until the variable has been instantiated. In [6]) it is achieved by using a new indexical term *val(X)* that delays the activation of a constraint in which it occurs until X has been instantiated. Here is the definition of the difference constraint:

```
'x≠y'(X,Y) :- X in -{val(Y)}, Y in -{val(X)}.
```

Such a suspension can be described in a concurrent constraint framework by using an *ask* mechanism ([13]). But, in our framework, the semantics of the computed result can be formalized also by using reduction rules, chaotic iterations and closures, even if some constraints remain suspended. Actually, an extension of our framework is unnecessary since we can provide particular cases of our general reduction rules (definition 3.1). Merely, the definition of these reduction rules uses *conditions*. For example, we can give the (monotonic) rule r of type $(\{dom(X), dom(Y)\}, dom(X))$ to define "X in -{val(Y)}":

$$\begin{array}{rcl} r(dom(X), dom(Y)) = X & \text{if} & |Y| > 1 \\ r(dom(X), dom(Y)) = X - \{y\} & \text{if} & Y = \{y\} \\ r(dom(X), dom(Y)) = \emptyset & \text{if} & Y = \emptyset \end{array}$$

A general definition of such *conditional rules* is an ongoing work that could be applied to Gnu-Prolog [7].

So, high-level user constraints can be given a semantics in terms of closure. But moreover, if every variable is instantiated, such a constraint also has an *intended logic semantics* and its definition has a declarative reading. For example, let us take the rule:

```
'x≥y'(X,Y) :- X in min(Y)..infinity, Y in 0..max(X).
```

with $dom(X) = \{x\}$ and $dom(Y) = \{y\}$. It has the obvious *intended semantics* $x \geq y$. Hence, the above closures are in general just *approximations* of these intended relations. On this basis, the framework of section 4, with variables and (concrete) constraints can be applied to the "X in r" scheme, mixing high-level user constraints with unary domain constraints (for each FD variable, it provides the unary constraint $x \in dom(X)$). The high-level constraints participate to the definition of the solution set while relaxation is only made on the unary domain constraints. The details and applications are outside the scope of the present paper.

5.4 Comparison with other works

In [5], Benhamou introduces narrowing operators that operate on approximate domains. The latters allow to represent continuous domains such as reals in a finite way. Then the computation only takes place in a large but finite domain upon which our formalism applies. Adequacy between the approximate and the intended domain is outside of this scope. This consideration made, the narrowing operators are similar to our relaxation operators, with the same expected properties. An interesting remark is the necessity for incomplete labeling of this setting due to the large size of the domains.

In [2, 3], Apt characterizes most consistency algorithms in a common setting as instance of chaotic iteration algorithms. The focus is made more on algorithms and definition of operators for various consistencies than on declarative characterization of the computed object. In [4], *rule-consistency* is introduced by rules that define the complementary of those of [10]. In [1], Apt propose to consider both relaxation and spliting rules, but the interaction between them is only sketched, having still in mind an algorithmic point of view.

6 Conclusion

We present in this paper a co-inductive semantics that models constraint solving including propagation and labeling. To achieve the latter point, we introduce new restriction rules that cut out the search space in order to explore it piece by piece. This formalism is general enough to model most existing propositions including the glass-box approach used for finite domain constraint solving in Gnu-Prolog.

As a semantics foundation, this work can be used as a basis for new developments in the domain of CSPs verification, debugging and more generally program analysis.

References

[1] K. R. Apt. A proof theoretic view of constraint programming. *Fundamenta Informaticae*, 34(3):295–321, June 1998.

[2] K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.

[3] K. R. Apt. The rough guide to constraint propagation. In *Constraint Programming CP'99*, 1999.

[4] K. R. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *Constraint Programming CP'99*, 1999.

[5] Frédéric Benhamou. Heterogeneous constraint solving. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming, 5th International Conference, ALP'96*, volume 1139 of *lncs*, pages 62–76, Aachen, Germany, 25–27 September 1996. Springer.

[6] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27(3):185–226, 1996.

[7] Daniel Diaz. *Gnu-Prolog Reference manual.* GNU, 1.1 edition, 2000.

[8] John W. Lloyd. *Foundations of logic programming.* Springer-Verlag, second edition, 1987.

[9] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[10] Ugo Montanari and Francesca Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.

[11] Ugo Montanari and Francesca Rossi. *Finite domain constraint solving and constraint logic programming*, chapter 11, pages 201–222. Logic Programming. MIT Press, 1993.

[12] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis.* Springer-Verlag, 1999.

[13] Vijay A. Saraswat. *Concurrent Constraint Programming.* ACM Doctoral Dissertation Awards. MIT Press, Cambridge, Massachusetts, 1993.

[14] E.Y. Shapiro. *Algorithmic program debugging.* ACM Distinguished Dissertations. ACM Press, 1982.

[15] M. H. van Emden. Value constraints in the CLP scheme. *Constraints*, 2:163–183, 1997.

[16] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). draft, 1991.