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/

Learning Interval Bounds of Indexical-based Solver

Michel Bergère, Thi Bich Hanh Dao, AbdelAli Ed-Dbali, Gérard Ferrand, Arnaud Lallouet, Andrei Legtchenko, Lionel Martin and Christel Vrain
Université d'Orléans, LIFO

Rapport N^o **2002-07**

Learning Interval Bounds of Indexical-based Solver

Michel Bergère, Thi Bich Hanh Dao, AbdelAli Ed-Dbali, Gérard Ferrand, Arnaud Lallouet, Andrei Legtchenko, Lionel Martin, Christel Vrain

Université d'Orléans — LIFO
BP 6759 — F-45067 Orléans cedex 2
`mailto:Arnaud.Lallouet@lifo.univ-orleans.fr`

Abstract. The pioneering works of [4] and [1] have shown that the construction of rule-based solvers could be automated using machine learning techniques. Both works implement the solver as a set of CHRs [10]. But many solvers use the more specialized chaotic iteration of operators [9, 3] as operational semantics and not CHR's rewriting semantics. In this paper, we first define a language-independent framework for operator learning and then we apply it to the learning of partial arc-consistency operators for a subset of the indexical language of Gnu-Prolog [7] and show the effectiveness of our approach by two implementations. On tested examples, Gnu-Prolog solvers are learned from their original constraints and powerful propagators are found for user-defined constraints.

Keywords : CSP; Consistency; Learning; Rule-based constraint solver.

1 Introduction

Building a constraint solver is notoriously a complex task [8, 14, 12] and it has been recently demonstrated that at least some parts of this conception can be automated by systematic search [4] or by using machine learning techniques [1, 2].

An efficient technique for computing consistencies in Constraint Satisfaction Problems (CSPs) is to use a data representation for the CSP and a set of operators whose common fixed-point models the expected consistency. The operators are then applied via chaotic iteration [9, 3] up to reach their common fixed-point. But solvers differ in the way they represent these operators: they can be written in the implementation language [14, 12] or in a higher-level language such as indexicals [16]. An indexical operator is written "`X in r`" where `X` is the name of a variable, and `r` is an expression which limits the range of possible values for `X` and which may depend on other variables' current domains.

For example, the constraint $and(X, Y, Z)$, defined by the following table, yields three indexical operators, one for each variable:

and :

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

generated indexicals:

```
X in min(Z) .. max(Z)*max(Y)+1-min(Y)
Y in min(Z) .. max(Z)*max(X)+1-min(X)
Z in min(X)*min(Y) .. max(X)*max(Y)
```

It is obvious that the expression of Z in function of X and Y is quite simple and intuitive whereas the expression of X in function of Y and Z is not. Building manually such expressions for arbitrary constraint is undoubtedly a challenge.

In this paper, we propose a framework to define the learning of such operators in three steps. We consider that a CSP is a set of n-ary constraints.

1. We define a notion of *semantic approximation* of a CSP which consists in replacing the original problem by an approximate one, from the point of view of the data-structure and the solutions. For a CSP C composed of several n-ary constraints, its approximation K uses different, usually simpler constraints (for example, the domain of each variable), and the notion of solution is replaced by the weaker one of consistency. Operators are introduced to transform these CSPs and their closure defines a consistent state.
2. We define a notion of *syntactic approximation*. The choice of a representation language, both for the data and the operators, is of crucial importance since the language may not allow to represent all possible data or operators. This defines a second level of approximation but also a concrete representation. For example, intervals are a data representation for variable domains and indexicals for operators.
3. It is then possible to define a learning space of all possible operators and check the adequation of each candidate operator towards the definition of the constraint. Operator learning consists in an exploration of this learning space to look for the best candidate.

We propose a framework and two implementations of indexical operator learning specialized to the context of the partial arc-consistency of Gnu-Prolog [7] which consists in contracting the bounds of the variables domain, represented as intervals. The learned operator can be intuitively thought as the best possible summary of the reductions done by the constraint on every possible interval which could be encountered during the solving process.

But many important theoretical and practical issues arise from this process: what is the quality of the learned solver ? Namely, is the solver *correct* ? What is its *completeness* ? And perhaps the most important: what is its pruning power ?

The paper is organized as follows. In section 2, preliminaries on CSP solving are given. In section 3, we present our approximation framework, from which the learning problem is derived. In section 4, we specialize our framework for the learning of indexicals for partial arc-consistency and we present the language biases we used. Section 5 is devoted to the implementations and section 6 gives an illustration on some examples.

2 Preliminaries

Let V be a set of variables and $D = (D_X)_{X \in V}$ their domains. The domain we consider are finite and totally ordered sets. For $W \subseteq V$, we denote by D^W the set of tuples on W, namely $\Pi_{X \in W} D_X$. Therefore, we have $\Pi D = D^V$. Projection of a tuple or a set of tuples on a set of variables is denoted by $|$, natural join of two sets of tuples is denoted by $\bowtie$.

Definition 1 (Constraint).
A constraint c *is a pair* (W, T) *where*

- $W \subseteq V$ *is the* arity *of the constraint* c *and is denoted by* $var(c)$*.*
- $T \subseteq D^W$ *is the* solution *of* c *and is denoted by* $sol(c)$*.*

Definition 2 (Approximation ordering).
A constraint $c' = (W', T')$ approximates $c = (W, T)$*, denoted by* $c \subseteq c'$*, if:*

$$var(c') = var(c) \quad and \quad sol(c) \subseteq sol(c')$$

Definition 3 (Join). *The* join *of* c *and* c' *is the constraint*

$$c \bowtie c' = (var(c) \cup var(c'), sol(c) \bowtie sol(c'))$$

Definition 4 (CSP). *A* CSP *is a set of constraints.*

Join is naturally extended to CSPs and a CSP C is identified to the join of its constraints $\bowtie C$. This defines the *solutions* of the CSP. A direct computation of this join is too expensive to be tractable. This is why various methods have been considered so far to compute the solutions of a CSP. We present here a short description of some of them which is useful for a comparison with other works (the presentation order is not significant):

- *search methods*: basically, it amounts to try values for the variables and test the constraints for satisfiability. It is one of the most studied methods and its refinements are countless. Almost all other methods are hybridized with search to get completeness.
- *symbolic transformations*: by this, we refer to a variety of syntactic transformations of the constraints in order to obtain a solved form. The *CLP*($\mathcal{R}$) [11] system obviously falls into this category since it uses the simplex method to transform the linear part of the constraint store. The solved form is obtained if the non-linear part eventually becomes linear, otherwise the constraints are simply returned to the user.
  The next level of abstraction comes with the CHR language [10] which allows to describe rewriting-based solvers in a simple and versatile way. It allows the definition of user predicates as well as rules to transform and simplify stores. A store is a set of syntactic atoms and rules which transform a store into a simpler one are either implications or equivalences. For example, here is a possible rule to simplify a boolean store:
  $$\texttt{and(X,Y,Z)} \wedge \texttt{Z=1} \iff \texttt{X=1} \wedge \texttt{Y=1} \wedge \texttt{Z=1}$$
  In our vision, the main characteristics of this language is that it uses first-order variables, or in other terms, that a variable represents an element of the domain.
- *approximations*: to be solved, constraints are represented, not only by their syntactic form, but also extensively by their solutions. A state is an over-approximating CSP and transitions from a CSP to a smaller one are represented by correct, monotonic and contracting operators which are iterated up to reach a property called *consistency*. Each consistency represents a notion of optimality in approximation. The contracting operators are most of the time written in the solver's implementation language (C++ for Ilog Solver [14], Claire for The Choco System [12], ...) and this is probably the reason why this formalism is often perseived as only implementation-relevant.
  There is however an exception with the "indexicals" language introduced in [16] and used in the Gnu-Prolog system [7] among others. In this framework, variables are second-order and designate constraints instead of domain elements. It allows to describe operators of the form "X in r" where X represents the domain of the variable, "in" is the set inclusion and r is an expression of some set language. We consider that this paradigm is more general than it may seem since it is parametrized by the language of r and deserves further research since there is still no formalism having the same generality and expressive power to define indexical operators as the CHR formalism for the symbolic transformation point of view. A step towards this goal may be the so-called delay clauses of B-Prolog [18]. Indeed, the choice of the "r" language is of particular importance in a learning perspective.
- *other methods*: various other methods (see for example [15]) have been tackled to solve CSPs, such as graph transformations, preconditioning, local search or metaheuristics. Their study is orthogonal to our concern.

Now we are more precise concerning approximations.

3 A model of approximation for CSPs

In this section, we provide a high-level model for the approximation made by consistency. The set-theoretic formulation we give of this model allows to describe very precisely the approximation process in a language-independent way. The basic idea is to clearly separate the CSP to be solved from its approximation (and more generally from the sequence of approximating CSPs).

3.1 Semantic approximations

Let C be a CSP over a set of variables V. Since a CSP is somehow identified with its join constraint $\bowtie C$, it can be approximated by a CSP K over the same set of variables V and such that $\bowtie C \subseteq \bowtie K$. In this case and by extension, we write $C \subseteq K$.

Intuitively, the CSP K is intended to be physically represented, for example (but not exclusively) by sets of tuples. This is why C and K may be built on completely different constraints. When all constraints in K are unary, and thus represent the domain of variables, it yields to the well-known "domain reduction scheme" used in most solvers. We call K the *approximating* CSP. Here is an example of two different approximating CSPs:

Example 5. Let C be the CSP composed of one constraint: $c = (\{X, Y, Z\}, \{(0,0,0), (0,0,1), (0,1,0), (1,0,1)\})$. Here are two approximating CSPs $K_1 = \{x, y, z\}$ and $K_2 = \{x, yz\}$ which are represented in figure 1. Here $x = (\{X\}, \{(0), (1)\})$ is an unary constraint representing the domain of X and $yz = (\{Y, Z\}, \{(0,0), (0,1), (1,0)\})$ is an arbitrary binary constraint. The constraints y and z are defined in the same straightforward way.

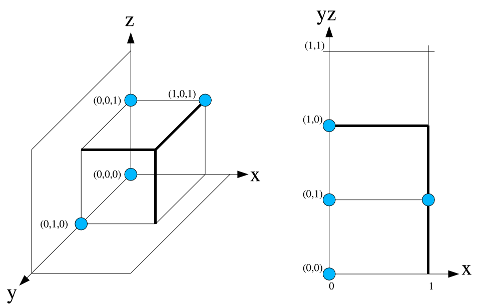

Fig. 1. Two different approximations for a CSP

Most of the time, switching from C to K provides a gain in terms of memory consumption. For example, in the domain reduction approximation for n variables and a domain of size m, it boils down from m^n to $m * n$. But the trade-off is that representable approximations are less precise since they are limited to (union of) cartesian products.

In the rest of the paper, C denotes the CSP to be solved, and K its approximating CSP. Moreover, we only use the domain reduction scheme. This means that the approximating CSP K is always composed of one unary constraint x per variable $X \in var(C)$ (lowercase letter designates the constraint while uppercase designates the variable). Since our goal is to find the best approximation (in some sense), we call an approximating CSP K a *search state* and the set of such states is the *search space* $SS = \Pi_{X \in V} \mathcal{P}(D_X)$. We just indicate that this framework can be extended to approximations made with constraints of arbitrary arity but domain approximations help to keep simplicity to the notations.

3.2 Consistencies

In order to reach a certain degree of precision, approximations have to be refined: a sequence of over-approximating CSPs $(K_i)_{i \in \mathbb{N}}$ such that $\forall i \geq 0, K_i \supseteq K_{i+1}$ is built, up to reach a closure called *consistency*. Transitions between states are expressed by operators. The nature of the operators determines the consistency as long as they meet the properties of correctness, monotony and contractance [3]. The important point is that they are defined according to some constraint $c \in C$ while their input is (some of) the constraints of the approximating CSP K. In order to limit the complexity of such operators, they can be classified by the number of constraints from C they need to be defined. Most of the time, an operator uses only one constraint of C to reduce a constraint of K as in the following example:

Example 6 (Projection operator). With the same CSP as example 5 we can define the projection operator, say on variable X. This operator transforms the CSP $K_1 = \{x, y, z\}$ in, say, $K_1' = \{x', y, z\}$ in which only the constraint x has been reduced into x' with $sol(x') = sol(x) \cap (K_1 \bowtie c)|_X$. Projection operators are used to define (hyper)arc-consistency.

A set of operators is associated to the CSP and is iterated until reaching their common closure. This is often done by chaotic iteration [9, 3]. We are concerned in this paper with the learning of such operators.

Let $c \in C$. A unary constraint $x \in K$ representing the domain of a variable $X \in var(c)$ is arc-consistent with respect to c if:

$$sol(x) = sol(K \bowtie c)|_X$$

A CSP K is arc-consistent if each of its constraints is. This immediately defines a notion of operator which can be fed into a chaotic iteration algorithm in order to obtain the consistency.

Since projection of the constraint solutions on a given variable only contains values which may be extended to a tuple satisfying the constraint, arc-consistency is the best monotonic approximation which can be obtained by considering only one constraint at a time. However, over-approximating operators with respect to arc-consistency may be used to define weaker consistencies. Their lesser pruning power is balanced by a lower computation time and a lower space representation cost. An example is the *partial arc-consistency* of Gnu-Prolog which represents variable domains only as intervals and thus only contracts bounds of the intervals.

A set of operators is associated to a CSP. Usually, to each constraint is associated an operator for each variable it contains. Let R be the set of reduction operators for a CSP C. We write $R \downarrow K$ the closure of K by the operators of R, i.e. the greatest CSP $K' \subseteq K$ such that $\forall r \in R, K' = r(K')$. A search state K is R-consistent if $K = R \downarrow K$.

3.3 Correctness and Completeness

Correctness for an operator r means that no solution is lost, or in other terms, that $C \subseteq K \Longrightarrow C \subseteq r(K)$. This is a strongly needed property since an incorrect solver may discard a solution for ever. Completeness, however, is not a required property at the operator's level since this is achieved by the search mechanism.

But often solvers require the operators to be *singleton complete*, i.e. complete only for singletons. Let us define this notion:

Definition 7 (Singletonic CSP).
An approximating CSP K is singletonic if $\forall k \in K$, $sol(k)$ is a singleton.

Intuitively, a singletonic CSP K on the set of variables V represents a tuple of D^V, a potential solution of the CSP C.

Definition 8 (Singleton Completeness).
Let C be a CSP and R its associated set of operators. The set R is singleton complete *if for any singletonic CSP K, representing the tuple t, we have*

$$t \in sol(C) \Longleftrightarrow K = R \downarrow K$$

This means that a tuple is solution if accepted by *all* operators while a non-solution tuple must be rejected (at least) by *one* operator.

Singleton completeness is the basis of the compilation scheme of Gnu-Prolog since the (high-level) constraints such as $X = Y + Z$ are replaced by an "equivalent" formulation in the indexical language. However, the compilation can be effective only if the operators are able to distinguish between a solution and a non-solution, at least at the tuple level. Operationnally, it means that, when dealing with a candidate solution, the consistency check can be realized by the propagation mechanism itself. But an important remark is that singleton completeness is a global property and is thus highly complex to check for an arbitrary set of operators because two non-solution tuples may be rejected by two different operators. In contrast, correctness check for an operator remains a local verification which only involves this operator.

3.4 Syntactic approximations: intervals and indexicals

The above formalism is language-independent but we still need to represent the domains and operators since learning is impossible without a representation. First, intervals are chosen for domain representation for their low space cost. This notion of approximation, different than consistency, comes from interval analysis and is presented for example in [3] where it is based on a family of subsets. This is an approximation of the domain while the consistency defines an approximation of the solutions. It defines the representable subsets of a variable's domain.

Then, how operators can be represented in a given language and then learned ? The indexical language introduced in [16] and used in Gnu-Prolog [7] is a possible answer. An operator is expressed by a second-order constraint $x \subseteq r$ where x is a constraint of the approximating CSP K and r a constraint expression in a given language. Since we consider that all constraints in K are unary, the evaluation of r is a "range", i.e. another monadic constraint.

Example 9. For example, the high-level constraint `X #= Y+C` is compiled for partial arc-consistency by Gnu-Prolog into two indexical operators:

`X in min(Y)+C .. max(Y)+C` and `Y in min(X)-C .. max(X)-C`

For full arc-consistency, the same constraint is compiled to:

`X in dom(Y)+C` and `Y in dom(X)-C`

If we restrict ourselves to the partial arc-consistency, the language defined in [6] has the following set constructors: "..", which denotes the middle of an interval whose bounds are specified by expressions in a term sub-language and "`&`", "`:`" and "`-`" which stand respectively for intersection, union and complementation. Moreover, the term sub-language allows the writing of various arithmetic operations. All of this makes a rather expressive language. A subset of the indexical language of Gnu-Prolog which defines the partial arc-consistency is recalled in figure 2. In contrast with the operator given in example 6 which uses the projection of the constraint, indexicals define an expression which may reduce a variable domain only in function of the constraint's other variables.

```
c  ::=   X in r
r  ::=   t1 .. t2                                        (interval)
         { t }                                           (singleton)
         r1 : r2                                         (union)
         r1 & r2                                         (intersection)
t  ::=   min(Y)                                          (indexical term min)
         max(Y)                                          (indexical term max)
         ct                                              (constant term)
         t1+t2 | t1-t2 | t1*t2 | t1/<t2 | t1/>t2         (integer operations)
ct ::=   n | infinity                                    (values)
         ct1+ct2 | ct1-ct2 | ct1*ct2 | ct1/<ct2 | ct1/>ct2
```

Fig. 2. A subset of Gnu-Prolog indexical language

Yet, and since the Gnu-Prolog indexical language does not contain conditional expressions, the formulation of some operators may be operationally inefficient. This is why Gnu-Prolog implements an ad-hoc delay mechanism by the syntax `val(X)` which represents the same value as `dom(X)` but delays until X is instanciated. It is the only guard of the language.

4 Description of the learning problem

In this section, we present the example space, the learning space and the learning algorithm. Semantic approximations of CSPs are a vast framework and many choices have to be made. Here we present as restrictions the main choices we made to get a tractable learning problem.

Let $c = (W, T) \in C$ be a constraint. Let us define the exact projection function of c on a variable $X \in W$. First, for $W \subseteq V$, let $SS_W = \Pi_{X \in W} \mathcal{P}(D_X)$ be the search space restricted to W. The projection function of c on X is defined as follow:

Definition 10 (Projection function).
Let C be a CSP, $c \in C$, $X \in var(c)$. The projection function of c on X is $c_X : SS_{var(c)-\{X\}} \rightarrow \mathcal{P}(D_X)$ and is defined by:

$$c_X(s) = \{t_X \in D_X \mid t \in sol(c) \wedge t|_{var(c)-\{X\}} \in s\}$$

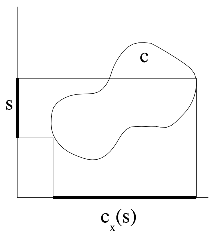

Fig. 3. Projection function

Our goal is to learn a function L^c_X which mimics the behavior of c_X and is at least correct. This means that $\forall s \in SS_{var(c)-\{X\}}, L^c_X(s) \supseteq c_X(s)$. But we also want to be the closest possible to the projection function since the constant function $s \mapsto D_X$ is correct but unuseful.

Definition 11 (Example set). *The function c_X, or equivalently the set of pairs $Ex(c, X) = \{(s, c_X(s)) \mid s \in SS_{var(c)-\{X\}}\}$ is called the* example set *for L^c_X.*

Learning is often made from an incomplete set of examples, and a part of the learning process consists in finding a suitable generalization. In solver learning, a missing example may yield an incorrect behavior, which contradicts the most crucially expected solver property.

Restriction 12. Since we want to get correct operators, we use the whole example set in the learning process.

The size of this example set is $2^{\Sigma_{X\in var(c)}|D_X|}$, which is in general far too large to be completely traversed. In contrast, the interval space has a size in $O(\Pi_{X\in var(c)}|D_X|^2)$. In order to shrink the search space, arbitrary subsets can be "rounded" to their next including interval.

Restriction 13. We compute with intervals.

Nevertheless, the output of an operator could be any subset. In order to further reduce the learning space, we set the following language bias:

Restriction 14. The output of the learned operator is a single interval.

Because of this restriction, the operator L_x^c is expressed in the indexical language by its minimal and maximal bounds $minL_x^c$ and $maxL_x^c$, which are arithmetic expressions not involving X. Note that the power of indexicals is not fully used but also that it is sufficient to represent the partial arc-consistency used in Gnu-Prolog.

Now we make these restrictions more formal in order to define the example set really used in the learning process. First, interval lattices are a particular case of lattice approximations:

Definition 15 (Lattice approximation). *Let E be a set and P a sub-lattice of $\mathcal{P}(E)$ such that $\emptyset \in P$ and $E \in P$. For $e \in \mathcal{P}(E)$, we define:*

$$up_P(e) = \inf\{p \in P \mid e \subseteq p\}$$

Let Int_X be the interval lattice of D_X. In order to define the input of the function to be learned, we need to define the search space in which the computation takes place. It consists of a unique interval for each variable. For $W \subseteq V$, let $Int_W = \Pi_{Y\in W} Int_Y$. The interval approximation can be extended to cartesian products by $up_{Int_W} : SS_W \to Int_W$ defined by:

$$s \mapsto \Pi_{Y\in W}\ up_{Int_Y}(s_Y)$$

By computing in the interval lattice, we preserve the correctness:

Proposition 16.

$$\forall s \in SS_{var(c)-\{X\}},\ c_X(s) \subseteq c_X(up_{Int_{var(c)-\{X\}}}(s))$$

Proof. By monotony of c_X.

Definition 17 (Approximate example set).
The approximate example set *for L_X^c is given by the function*

$$Ex^{Int}(c,X) : Int_{var(c)-\{X\}} \to Int_X$$

such that

$$Ex^{Int}(c,X)(s) = up_{Int_X}(c_X(s)))$$

The bounds of L^c_X can be learned separately from a separate example set. For example, for $minL^c_X$, the example set is given by the function $Int_{var(c)-\{X\}} \rightarrow \mathbb{N}$ defined by:

$$Ex^{Int}_{min}(c, X) : s \mapsto min(Ex^{Int}(c, X)(s))$$

In the following, we focus on $minL^c_X$, but the other bound is obtained by the same method.

Now, we cannot be more precise without introducing a language to express operators. The remaining problem is to find a suitable expression in the sub-grammar of terms and constant terms (entries t and ct in figure 2) which follows as closely as possible the behavior of the function $Ex^{Int}_{min}(c, X)$. In order to limit the complexity of the learning process, we use the following language bias. Since the sum is present in the language, our technique consists in fixing the general form of the function to be learned to a linear combinaison of terms, each one being expressed without the addition. Let $\mathcal{L}$ be the term language on the following sub-grammar of indexicals:

t ::=	`min(Y)`	(indexical term *min*)
	`max(Y)`	(indexical term *max*)
	$\mathtt{t_1{*}t_2}$	(integer operations)

Now let $\mathcal{L}_2$ be the sub-language of $\mathcal{L}$ limited to the expressions whose depth is lesser than 2.

Example 18. With $V = \{X, Y\}$, $\mathcal{L}_2 = \{$ `min(X)`, `min(Y)`, `max(X)`, `max(Y)`, $\mathtt{min(X)}^2$, $\mathtt{min(Y)}^2$, $\mathtt{max(X)}^2$, $\mathtt{max(Y)}^2$, `min(X)*min(Y)`, `min(X)*max(Y)`, $\ldots\}$

Then we can express $minL^c_X$ as the following linear form:

$$minL^c_x = \Sigma_{w \in \mathcal{L}_2} \alpha_w * w + \alpha_0$$

The last part of the learning algorithm consists in finding the coefficients α_w, in order to meet the correction constraints. For each example s, the value of $minL^c_X(s)$ is correct if lesser than the real value $Ex^{Int}_{min}(c, X)(s)$.

In contrast, completeness cannot be ensured the same way because it would sometimes prevent the system to have a solution. Singleton completeness, which is not a local property, cannot be expressed at this level. Our approach consists in finding the most complete function in a certain sense, by minimizing a quality function for each candidate. We tried different quality functions and the results are presented in the next section. A typical quality function is to minimize the global error between the candidate and the real value. Hence we get the following linear program:

`minimize:`

$$\Sigma_{s \in Int_{var(c)-\{X\}}} Ex^{Int}_{min}(c, X)(s) - minL^c_X(s)$$

`subject to:`

$$\forall s \in Int_{var(c)-\{X\}} \quad , \quad [\Sigma_{w \in \mathcal{L}_2} \alpha_w * w + \alpha_0](s) \ \leq \ min(up_{Int}(c_X(s)))$$

The correctness of the algorithm is ensured by construction.

One may wonder if the restrictions we made and the biases we used do not yield a too much restricted framework. Here are some answers:

- The formalism of semantic approximations we use is very general and allow us to test many hypotheses. Indeed, with our definition of semantic approximation, all kinds of consistencies can be formalized in a language independent way. Indexicals are however high-level enough to allow easily the handling of the many syntactic transformations necessary to the learning process.

- The results we present in section 6 show that the operator learned are of good quality. In particular on regular constraints, we are able to find the classical operators of Gnu-Prolog. On arbitrary constraints, we still obtain a good pruning power.
- The choices we made present a good balance between the complexity of the learning task and the size of the explored search space. In particular, the use of a strong language bias is a common feature in machine learning [17].
- The complete construction of a solver may probably not be fully automatic. But a collection of learning tools with different techniques and algorithm may be of great help. We rather think of these tools as being a solver design environment.

5 Implementations

We have realized two systems implementing this framework. A first system handles constraints of arbitrary arity, uses the simplex algorithm and yields X in r in the form described above (a linear combinaison of elements of $\mathcal{L}_2$) and the second one handles only binary constraints, uses a genetic algorithm, and propose three different (but sill fixed) forms of operators.

5.1 The Simplex learner

The principle of this implementation is to solve the previous linear programming problem with the simplex algorithm: the simplex learner is a C program which calls the solver lp_solve[1].

The command-line user interface allow mainly to build minimal and maximal bounds for each variable with basic simplifications and to compute reductions obtains with these bounds.

5.2 The Genetic learner: GA-ILearner

This second implementation uses genetic algorithms to improve the quality of a population of candidate solutions. This C++ application has a graphic interface which allows the user to "draw" his constraint with the mouse. The user constraint appears on screen as red dots (see figure 4). Only binary constraints are allowed in order to be representable on screen, but variable domains range up to 50×50. The user can learn the bounds of the functions separately, or specify them manually. Three fixed forms of functions are proposed for each bound:

- linear:

$$X_{min} \text{ in } A * min(Y) + B * max(Y) + C$$

- rational (an example of rational approximation is given in figure 4):

$$X_{min} \text{ in } \frac{A}{min(Y)+1} + \frac{B}{max(Y)+1} + C$$

- quadratic (an example of quadratic approximation is given in figure 5):

$$X_{min} \text{ in } A*min(Y)^2+B*max(Y)^2+C*min(Y)*max(Y)+D*min(Y)+E*max(Y)+F$$

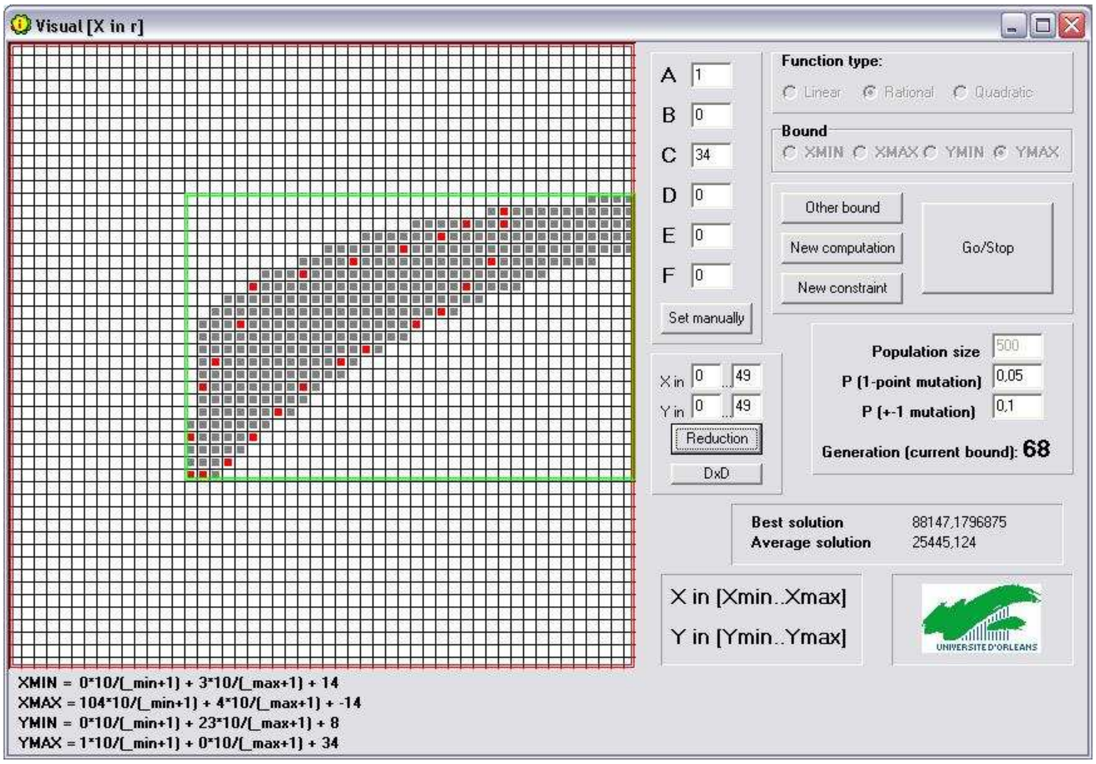

Fig. 4. Rational approximation of a cloud

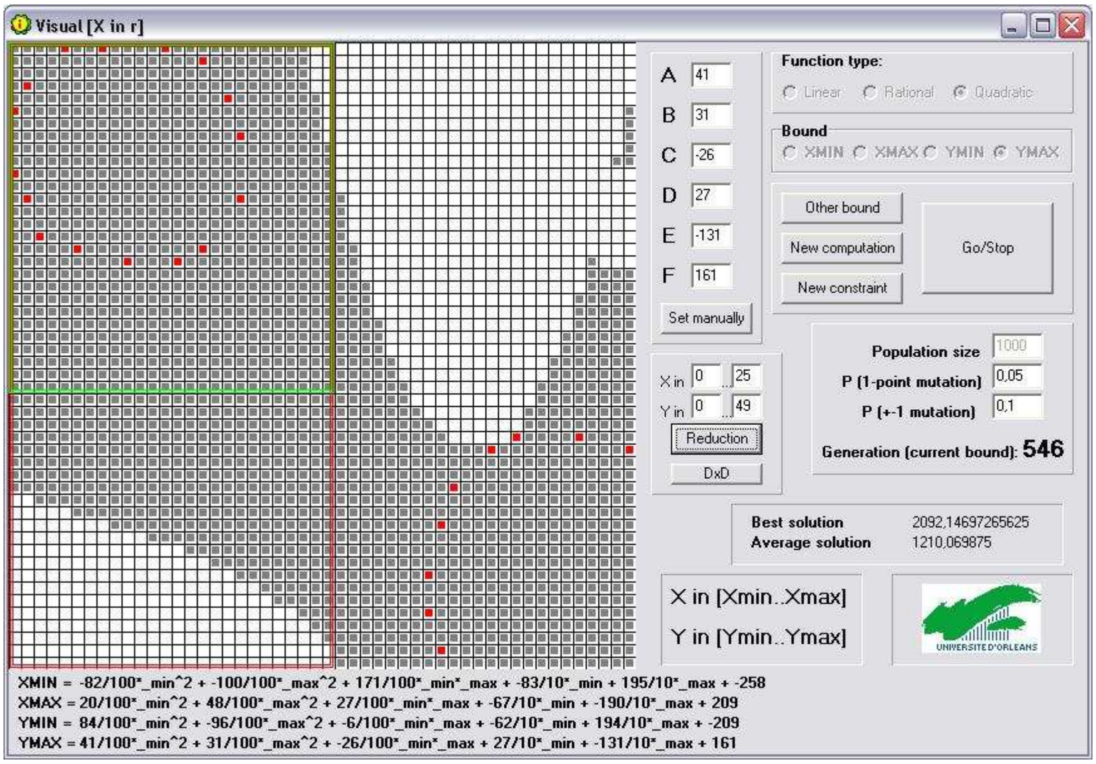

Fig. 5. Quadratic approximation of a disconnected cloud

The shape of the singletons which satisfy the constraint appears on screen as grey dots. The user may then specify an interval, which appears as a red square, and try the reduction given by his/her indexicals, which appears as a green square (see figure 5). Since the learned indexicals may not be monotonic, the reduced interval may be smaller than the singleton solutions. However, correctness with respect to the constraint's solutions is ensured by construction.

The construction of the indexical is left to a genetic algorithm in which individuals are a sequence of bits representing the coefficients. The length of an individual depends on the chosen function form. The quality function is inversely proportional to the square of the error between the candidate and the example set. We tried as definition of error the simple sum of the individual errors on each intervals and the sum of individual errors relative to the size of the interval. Incorrect individuals are strongly penalized. Surprisingly, these notions of error, besides the complexity of their evaluation, exhibit a relatively flat search space with very deep and narrow local minimas. This sometimes cause a slow convergence of the population, which is ordinary of a few seconds.

The rest of the algorithm is classical: n-points cross-over and two kinds of mutations (increment and random). Note that it may seem strange to use a genetic algorithm for linear optimization but this algorithm is a better candidate to move towards genetic programming which would allow the search of the function form in addition to the coefficients.

5.3 Analysis of the results

By using these tools, we obtained the following results:

- Rational functions have an excellent pruning power which is often better than linear ones. This result is purely experimental and is based on our tests.
- Singleton completeness is difficult (and sometimes impossible) to ensure. However, this property is only needed to justify the compilation scheme. If we have a way to check the satisfiability of a constraint, we may freely use these indexicals. Moreover, they may yield more pruning than singleton complete ones, thus building a faster solver.
- Monotony of the indexical expression is not needed nor desirable since it may prevent interesting but non-monotonic functions to be discovered. For finite domains, this property can be relaxed because the correctness and contractance are enough for the sequence to converge.

6 Examples

We present in this section some examples of learned indexicals, illustrated by reductions using them. In the following, the notation $\texttt{reduce}(m_1, M_1, m_2, M_2, \ldots, m_n, M_n)$ is used as a shorthand for the application of the n learned operators for the n variables of the constraint up to their common closure. For each variable i, m_i's and M_i's are integers defining the variable domain (the interval $[m_i, M_i]$).

The result we present in this section have been obtained with the simplex indexical learner implementation.

6.1 Boolean constraints

Primitive constraints

[1] ftp://ftp.es.ele.tue.nl/pub/lp_solve ©

- **And** With the conjunction constraint `and(X0,X1,X2)` representing the relation $X_2 = X_0 \wedge X_1$, our algorithm returns the following indexicals:

```
X0 in [  Min(X2) ..  1.00 - Min(X1) + Min(X1)*Max(X2) ]
X1 in [  Min(X2) ..  1.00 - Min(X0) + Min(X0)*Max(X2) ]
X2 in [  Min(X0)*Min(X1) .. Max(X0)*Max(X1) ]
```

These generated indexicals are equivalent to the ones given in [6]. The subtle difference is in favor of our expression since the result remains in $[0,1]$. Since domain are booleans and since an intersection is made with the evaluated expression, Gnu-Prolog indexical is still singleton complete. With $X_0 = 0$, the elements of this constraint are $(0,0,0)$ and $(0,1,0)$, and the reduction using these indexicals returns exactly these intervals:

```
-> reduce(0,0,0,1,0,1)

X0 in [  0.00 ..  0.00 ]
X1 in [  0.00 ..  1.00 ]
X2 in [  0.00 ..  0.00 ]
```

In the same manner, the only element of this constraint with $X_2 = 1$ is $(1,1,1)$ and the reduction returns that when $X_2 = 1$ then X_0 and X_1 must be 1 too:

```
-> reduce(0,1,0,1,1,1)

X0 in [  1.00 ..  1.00 ]
X1 in [  1.00 ..  1.00 ]
X2 in [  1.00 ..  1.00 ]
```

These indexicals are able to reject some non-solutions, like in the following reduction, where in the result, the second and third interval are empty:

```
->reduce(0,1,0,0,1,1)

X0 in [  1.00 ..  1.00 ]
X1 in [  1.00 ..  0.00 ]
X2 in [  1.00 ..  0.00 ]
```

- **Or.** For the constraint or `or(X0,X1,X2)`, our algorithm generates the following indexicals, which are also the same as in [6]:

```
X0 in [  Min(X2) - Min(X2)*Max(X1) ..  Max(X2) ]
X1 in [  Min(X2) - Min(X2)*Max(X0) ..  Max(X2) ]
X2 in [  Min(X0) + Min(X1) - Min(X0)*Min(X1) ..  Max(X0) + Max(X1) -
         Max(X0)*Max(X1) ]
```

- **Xor.** For the constraint exclusive-or `xor(X0,X1,X2)`, our algorithm generates the following indexicals:

```
X0 in [  Min(X1) +  Min(X2) - Min(X1)*Max(X2) - Min(X2)*Max(X1)
      ..  Max(X1) +  Max(X2) - Min(X1)*Min(X2) - Max(X1)*Max(X2) ]
X1 in [  Min(X0) +  Min(X2) - Min(X0)*Max(X2) - Min(X2)*Max(X0)
      ..  Max(X0) +  Max(X2) - Min(X0)*Min(X2) - Max(X0)*Max(X2) ]
X2 in [  Min(X0) +  Min(X1) - Min(X0)*Max(X1) - Min(X1)*Max(X0)
      ..  Max(X0) +  Max(X1) - Min(X0)*Min(X1) - Max(X0)*Max(X1) ]
```

The only element where $X_0 = 1$ and $X_2 = 1$ is $(1,0,1)$ and the only element where $X_0 = 0$ and $X_2 = 1$ is $(0,1,1)$. These indexicals imply also that when $X_0 = 1$ and $X_2 = 1$, then X_1 must be 0:

```
->reduce(1,1,0,1,1,1)

X0 in [  1.00 ..  1.00 ]
X1 in [  0.00 ..  0.00 ]
X2 in [  1.00 ..  1.00 ]
```

and when $X_0 = 0$ and $X_2 = 1$, then X_1 must be 1:

```
->reduce(0,0,0,1,1,1)

X0 in [  0.00 ..  0.00 ]
X1 in [  1.00 ..  1.00 ]
X2 in [  1.00 ..  1.00 ]
```

And all reductions with two variables fixed give the exact intervals.

Full adder
We consider the constraint `fulladder(X0,X1,X2,X3,X4)`, meaning $X_0 + X_1 + X_2 = 2 * X_3 + X_4$ and represented in figure 6. The generated indexicals are:

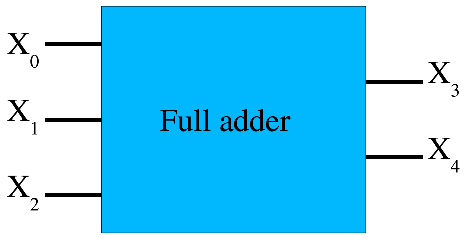

Fig. 6. The full adder

```
X0 in [  2.00.Min(X3) + Min(X3)*Min(X4) - Min(X3)*Max(X1) -
         Min(X3)*Max(X2)
      .. - Min(X1) - Min(X2)+  Max(X3)+  Max(X4) + Min(X1)*Max(X3) +
         Min(X2)*Max(X3) - Max(X3)*Max(X4) ]
X1 in [  2.00.Min(X3) + Min(X3)*Min(X4) - Min(X3)*Max(X0) -
         Min(X3)*Max(X2)
      .. - Min(X0) - Min(X2) + Max(X3) + Max(X4) + Min(X0)*Max(X3) +
         Min(X2)*Max(X3) - Max(X3)*Max(X4) ]
X2 in [  2.00.Min(X3) + Min(X3)*Min(X4) - Min(X3)*Max(X0) -
         Min(X3)*Max(X1)
      .. - Min(X0) - Min(X1) + Max(X3) +  Max(X4) + Min(X0)*Max(X3) +
         Min(X1)*Max(X3) - Max(X3)*Max(X4) ]
X3 in [  Min(X0) - Min(X1) +  Min(X2) + Min(X0)*Min(X1) -
         Min(X0)*Min(X2) - Min(X0)*Max(X4) + Min(X1)*Min(X2) +
         Min(X1)*Max(X4) - Min(X2)*Max(X4)
      .. 2.00.Max(X2) - Min(X4) + Min(X4)*Max(X0) + Min(X4)*Max(X1) -
         Min(X4)*Max(X2) + Max(X0)*Max(X1) - Max(X0)*Max(X2) -
         Max(X1)*Max(X2) ]
X4 in [  Min(X0) + Min(X1) + Min(X2) - Min(X0)*Max(X3) -
         Min(X1)*Max(X3) - Min(X2)*Max(X3)
      .. 1.00 - 3.00.Min(X3) + Min(X3)*Max(X0) + Min(X3)*Max(X1) +
         Min(X3)*Max(X2) ]
```

Let $X_0 = 0$ and $X_3 = 1$ and let us reduce the other variable's domains:

```
->reduce(0,0,0,1,0,1,1,1,0,1)

X0 in [  0.00 ..  0.00 ]
X1 in [  1.00 ..  1.00 ]
X2 in [  1.00 ..  1.00 ]
X3 in [  1.00 ..  1.00 ]
X4 in [  0.00 ..  0.00 ]
```

The reduction then says that in this case, X_1 and X_2 must be 1 and X_4 must be 0, which is exactly the corresponding element of this constraint.

6.2 Three-valued logic

Variables in three-valued logic can have one of the values 1, 0 and 0.5, which stand for true, false and unknown respectively. We present here the equivalence and the conjunction constraints.

- **Equiv**$_3$ The equivalence constraint `equiv3(X0,X1,X2)` represents the relation $X_2 = (X_0 \equiv X_1)$. The elements of `equiv3` are given in figure 7. For this

equiv$_3$:

X_0	X_1	X_2
0	0	1
0	0.5	0.5
0	1	0

X_0	X_1	X_2
0.5	0	0.5
0.5	0.5	0.5
0.5	1	0.5

X_0	X_1	X_2
1	0	0
1	0.5	0.5
1	1	1

Fig. 7. The constraint *equiv*$_3$.

constraint our algorithm returns:

```
X0 in [  1.00  - Min(X1) - 2.00.Max(X1) - Max(X2) + Min(X1)*Min(X1) +
         Min(X1)*Min(X2) + Max(X1)*Max(X1) + Max(X1)*Max(X2)  ..
         1.00 +  Max(X1) - Min(X2) - Min(X1)*Min(X1) +
         Min(X1)*Max(X2) + Min(X2)*Max(X1) - Max(X1)*Max(X1) ]
X1 in [  1.00  - Min(X0) - 2.00.Max(X0) - Max(X2) +
         Min(X0)*Min(X0) + Min(X0)*Min(X2) + Max(X0)*Max(X0) +
         Max(X0)*Max(X2)  ..
         1.00 +  Max(X0) - Min(X2) - Min(X0)*Min(X0) + Min(X0)*Max(X2) +
         Min(X2)*Max(X0) - Max(X0)*Max(X0) ]
X2 in [  1.00  - Max(X0) - Max(X1) + Min(X0)*Min(X1) + Max(X0)*Max(X1)
      ..  1.00  - Min(X0) - Min(X1) + Min(X0)*Max(X1) + Min(X1)*Max(X0) ]
```

When $X_0 = 0$ and $X_1 = 1$, these indexicals imply that $X_2 = 0$, which form exactly the corresponding element of the constraint:

```
->reduce(0,0,1,1,0,1)
X0 in [  0.00 ..  0.00 ]
X1 in [  1.00 ..  1.00 ]
X2 in [  0.00 ..  0.00 ]
```

And when $X_0 = 0$ and $X_2 = 0.5$, then X_1 must be 0.5, which is also an exact solution:

```
->reduce(0,0,0,1,0.5,0.5)

X0 in [  0.00 ..  0.00 ]
X1 in [  0.50 ..  0.50 ]
X2 in [  0.50 ..  0.50 ]
```

– **And**$_3$ Let us now consider the conjunction constraint `and3(X0,X1,X2)`, which is defined in figure 8. From this constraint our algorithm generates the following

and$_3$:

X_0	X_1	X_2
0	0	0
0	0.5	0
0	1	0

X_0	X_1	X_2
0.5	0	0
0.5	0.5	0.5
0.5	1	0.5

X_0	X_1	X_2
1	0	0
1	0.5	0.5
1	1	1

Fig. 8. The constraint and_3.

indexicals:

```
X0 in [ Min(X2) ..  1.00 -  Min(X1)*Min(X1) + Min(X1)*Max(X2) ]
X1 in [ Min(X2) ..  1.00 -  Min(X0)*Min(X0) + Min(X0)*Max(X2) ]
X2 in [ Min(X0)*Min(X1) ..  Max(X0) - Max(X0)*Max(X0) + Max(X0)*Max(X1) ]
```

The reduction with $X_0 = 0$ and $X_2 = 0.5$ gives:

```
->reduce(0,0,0,1,0.5,0.5)

X0 in [  0.50 ..  0.00 ]
X1 in [  0.50 ..  1.00 ]
X2 in [  0.50 ..  0.00 ]
```

We notice that the first and the third indexicals represent the empty interval, which mean that all elements whose $X_0 = 0$ and $X_2 = 0.5$ are rejected by these indexicals.
When $X_2 = 0.5$, these indexicals imply that X_0 and X_1 must not be 0, and when $X_2 = 1$, then X_0 and X_1 must be 1:

```
->reduce(0,1,0,1,0.5,0.5)

X0 in [  0.50 ..  1.00 ]
X1 in [  0.50 ..  1.00 ]
X2 in [  0.50 ..  0.50 ]

->reduce(0,1,0,1,1,1)

X0 in [  1.00 ..  1.00 ]
X1 in [  1.00 ..  1.00 ]
X2 in [  1.00 ..  1.00 ]
```

6.3 Sign of multiplication

The constraint `sign(X_0,X_1,X_2)` represents the relation where X_0, X_1 and X_2 are respectively the sign of the two operands and the one of the multiplication result. The variables can have the value -1 (negative), 0 (zero) and 1 (positive). Our algorithm yields the following indexicals:

```
X0 in [ -1.00 + 0.50.Min(X1)*Max(X1) + 0.50.Min(X1)*Max(X2) +
        0.50.Min(X2)*Max(X1) + 0.50.Min(X2)*Max(X2)
     .. 1.00 + 0.50.Min(X1)*Min(X2) - 0.50.Min(X1)*Max(X1) -
        0.50.Min(X2)*Max(X2) + 0.50.Max(X1)*Max(X2) ]
X1 in [ -1.00 + 0.50.Min(X0)*Max(X0) + 0.50.Min(X0)*Max(X2) +
        0.50.Min(X2)*Max(X0) + 0.50.Min(X2)*Max(X2)
     .. 1.00 + 0.50.Min(X0)*Min(X2) - 0.50.Min(X0)*Max(X0) -
        0.50.Min(X2)*Max(X2) + 0.50.Max(X0)*Max(X2) ]
X2 in [ 0.50.Min(X0) - 0.50.Max(X0) + 0.50.Min(X1) - 0.50.Max(X1) +
        0.50.Min(X0)*Min(X1) + 0.50.Max(X0)*Max(X1)
     .. -0.50.Min(X0) + 0.50.Max(X0) - 0.50.Min(X1) + 0.50.Max(X1)+
        0.50.Min(X0)*Max(X1) + 0.50.Min(X1)*Max(X0) ]
```

We obtain fine reductions when the values of two of the variables are fixed:

```
->reduce(0,0,0,0,-1,1)

X0 in [  0.00 ..  0.00 ]
X1 in [  0.00 ..  0.00 ]
X2 in [  0.00 ..  0.00 ]

->reduce(1,1,-1,1,0,0)

X0 in [  1.00 ..  1.00 ]
X1 in [  0.00 ..  0.00 ]
X2 in [  0.00 ..  0.00 ]

->reduce(1,1,1,1,-1,1)

X0 in [  1.00 ..  1.00 ]
X1 in [  1.00 ..  1.00 ]
X2 in [  1.00 ..  1.00 ]
```

However, there are cases when only one variable is fixed, there is no reductions:

```
->reduce(0,0,-1,1,-1,1)

X0 in [  0.00 ..  0.00 ]
X1 in [ -1.00 ..  1.00 ]
X2 in [ -1.00 ..  1.00 ]
```

6.4 All-different constraints

All-different is one of the most famous global constraint, which forces all of its variables to take different values. We present here examples of learned indexicals from different instances of this constraint. It is interesting to consider that while this constraint owns a special powerful propagation algorithm, Gnu-Prolog does not implement it and uses the simple translation to different ($\neq$) constraints. According to the compilation scheme of Gnu-Prolog, these constraints are delayed until one variable becomes ground. In many cases, our learned indexicals yield a better pruning.

- **Two-variables constraint on three-valued domains**
  The elements of this constraint are all possible combination (x_0, x_1) where $x_0, x_1 \in \{0, 1, 2\}$ and $x_0 \neq x_1$. Our algorithm returns the following indexicals:

```
X0 in [  0.0   ..  2.00 + 0.50.Min(X1) - 0.50.Min(X1)*Min(X1) ]
X1 in [  1.00 - 1.50.Max(X0) + 0.50.Max(X0)*Max(X0)
      ..  2.00 + 0.50.Min(X0) - 0.50.Min(X0)*Min(X0) ]
```

When $X_0 = 0$, then X_1 must be different from 0, that is X_1 may be 1 or 2. This case is represented by the following reduction:

```
->reduce(0,0,0,2)

X0 in [  0.00 ..  0.00 ]
X1 in [  1.00 ..  2.00 ]
```

Since with the form of indexical of partial arc-consistency, we can only reduce the minimal or maximal bound, when $X_0 = 1$, the reduction of X_1 is not representable:

```
->reduce(1,1,0,2)

X0 in [  1.00 ..  1.00 ]
X1 in [  0.00 ..  2.00 ]
```

– **Three-variables constraint on three-valued domains**
This 3-ary constraint has as elements all possible combination (x_0, x_1, x_2) where $x_0, x_1, x_2 \in \{0, 1, 2\}$ and each one of them is different from the others. The learned indexicals are:

```
X0 in [  4.00 - 2.00.Max(X1) - 2.00.Max(X2) + Max(X1)*Max(X2)
      ..  2.00 + 0.50.Min(X1) + 0.50.Min(X2) - 0.50.Min(X1)*Min(X1) -
          0.50.Min(X1)*Min(X2) - 0.50.Min(X2)*Min(X2) ]
X1 in [  4.00 - 2.00.Max(X0) - 2.00.Max(X2) + Max(X0)*Max(X2)
      ..  2.00 + 0.50.Min(X0) + 0.50.Min(X2) - 0.50.Min(X0)*Min(X0) -
          0.50.Min(X0)*Min(X2) - 0.50.Min(X2)*Min(X2) ]
X2 in [  4.00 - 2.50.Max(X0) - 2.50.Max(X1) + 0.50.Max(X0)*Max(X0) +
          0.50.Max(X0)*Max(X1) + 0.50.Max(X1)*Max(X1)
      ..  2.00 + 0.50.Min(X0) + 0.50.Min(X1) - 0.50.Min(X0)*Min(X0) -
          0.50.Min(X0)*Min(X1) - 0.50.Min(X1)*Min(X1) ]
```

We know that when $0 \leq X_0, X_1 \leq 1$, then X_2 must be 2. Since our algorithm is not complete, it does not return the only value 2 for the domain of X_2, nevertheless it removes 0 from this domain:

```
->reduce(0,1,0,1,0,2)

X0 in [  0.00 ..  1.00 ]
X1 in [  0.00 ..  1.00 ]
X2 in [  1.00 ..  2.00 ]
```

And because of the incompleteness, some of the cases where no solution exists cannot be detected, for example the following case, where the reduction returns $X_0 = X_1 = X_2 = 1$:

```
->reduce(0,1,0,1,0,1)

X0 in [  1.00 ..  1.00 ]
X1 in [  1.00 ..  1.00 ]
X2 in [  1.00 ..  1.00 ]
```

However, when X_0 is fixed to the maximal or minimal bounds, we still have quite fine reductions:

```
->reduce(0,0,0,2,0,2))

X0 in [  0.00 ..  0.00 ]
X1 in [  0.00 ..  2.00 ]
X2 in [  1.00 ..  2.00 ]

->reduce(2,2,0,2,0,2)

X0 in [  2.00 ..  2.00 ]
X1 in [  0.00 ..  1.00 ]
X2 in [  0.00 ..  1.00 ]
```

– **Four-variables constraint on four-valued domains**
  In this 4-ary constraint, variables take values from the domain $\{0, 1, 2, 3\}$. Its elements are all possible combinations (x_0, x_1, x_2, x_3), where $x_i \neq x_j$ for $i \neq j$. Our algorithm returns the following indexicals:

```
X0 in [  9.00 - 2.29.Max(X1) - 2.29.Max(X2) - 2.29.Max(X3) +
         0.43.Max(X1)*Max(X2) + 0.43.Max(X1)*Max(X3) + 0.43.Max(X2)*Max(X3)
      .. 3.00 + 0.29.Min(X1) + 0.29.Min(X2) + 0.29.Min(X3) -
         0.43.Min(X1)*Min(X2) - 0.43.Min(X1)*Min(X3) - 0.43.Min(X2)*Min(X3) ]
X1 in [  9.00 - 2.29.Max(X0) - 2.29.Max(X2) - 2.29.Max(X3) +
         0.43.Max(X0)*Max(X2) + 0.43.Max(X0)*Max(X3) + 0.43.Max(X2)*Max(X3)
      .. 3.00 + 0.29.Min(X0) + 0.29.Min(X2) + 0.29.Min(X3) -
         0.43.Min(X0)*Min(X2) - 0.43.Min(X0)*Min(X3) - 0.43.Min(X2)*Min(X3) ]
X2 in [  9.00 - 2.29.Max(X0) - 2.29.Max(X1) - 2.29.Max(X3) +
         0.43.Max(X0)*Max(X1) + 0.43.Max(X0)*Max(X3) + 0.43.Max(X1)*Max(X3)
      .. 3.00 + 0.29.Min(X0) + 0.29.Min(X1) + 0.29.Min(X3) -
         0.43.Min(X0)*Min(X1) - 0.43.Min(X0)*Min(X3) - 0.43.Min(X1)*Min(X3) ]
X3 in [  9.00 - 2.29.Max(X0) - 2.29.Max(X1) - 2.29.Max(X2) +
         0.43.Max(X0)*Max(X1) + 0.43.Max(X0)*Max(X2) + 0.43.Max(X1)*Max(X2)
      .. 3.00 + 0.29.Min(X0) + 0.29.Min(X1) + 0.29.Min(X2) -
         0.43.Min(X0)*Min(X1) - 0.43.Min(X0)*Min(X2) - 0.43.Min(X1)*Min(X2) ]
```

  When $X_0 = 0$, $0 \leq X_1 \leq 1$, $0 \leq X_2 \leq 2$, these indexicals allow to find the singletonic solution, where $X_0 = 0$, $X_1 = 1$, $X_2 = 2$ and $X_3 = 3$:

```
->reduce(0,0,0,1,0,2,0,3)

X0 in [  0.00 ..  0.00 ]
X1 in [  1.00 ..  1.00 ]
X2 in [  2.00 ..  2.00 ]
X3 in [  3.00 ..  3.00 ]
```

  And when $0 \leq X_0 \leq 1$ and $0 \leq X_1, X_2 \leq 2$, they remove the value 0 from the X_2's domain. This example shows that our learned function is able to perform more pruning than the algorithm in Gnu-Prolog:

```
->reduce(0,1,0,2,0,2,0,3)

X0 in [  0.00 ..  1.00 ]
X1 in [  0.00 ..  2.00 ]
X2 in [  0.00 ..  2.00 ]
X3 in [  1.00 ..  3.00 ]
```

6.5 Directed graphs

Let us introduce the following directed graph:

$$
\begin{tikzcd}
0,3 \arrow[r] \arrow[d] & 1,3 \arrow[r] & 2,3 \arrow[r] \arrow[d] & 3,3 \\
0,2 \arrow[d] & 1,2 \arrow[l] \arrow[u] & 2,2 \arrow[l] \arrow[d] & 3,2 \arrow[l] \arrow[u] \\
0,1 \arrow[r] \arrow[d] & 1,1 \arrow[r] \arrow[u] & 2,1 \arrow[r] \arrow[d] & 3,1 \arrow[u] \\
0,0 & 1,0 \arrow[l] \arrow[u] & 2,0 \arrow[l] & 3,0 \arrow[l] \arrow[u]
\end{tikzcd}
$$

We consider the constraint whose elements are (x_0, x_1, x_2, x_3), where $x_i \in \{0, 1, 2, 3\}$ and there is an edge from (x_0, x_1) to (x_2, x_3). With this constraint, our algorithm results:

```
X0 in [  -0.33.Min(X2) + 0.33.Min(X2)*Min(X2)
      ..  1.00 + 1.67.Max(X2) - 0.33.Max(X2)*Max(X2) ]
X1 in [  -0.33.Min(X3) + 0.33.Min(X3)*Min(X3)
      ..  1.00 + 1.17.Min(X3) + Max(X3) + 0.17.Min(X3)*Min(X3) -
          0.67.Min(X3)*Max(X3) ]
X2 in [  -0.33.Min(X0) + 0.33.Min(X0)*Min(X0)
      ..  1.00 + 1.67.Max(X0) - 0.33.Max(X0)*Max(X0) ]
X3 in [  -0.33.Min(X1) + 0.33.Min(X1)*Min(X1)
      ..  1.00 + 1.67.Max(X1) - 0.33.Max(X1)*Max(X1) ]
```

We see that from $(1, 1)$ there are two edge to $(2, 1)$ and $(1, 2)$, that is $X_2 \in [1, 2]$ and $X_3 \in [1, 2]$. The result of the reduction using these indexicals is quite close:

```
->reduce(1,1,1,1,0,3,0,3)

X0 in [  1.00 ..  1.00 ]
X1 in [  1.00 ..  1.00 ]
X2 in [  0.00 ..  2.00 ]
X3 in [  0.00 ..  2.00 ]
```

as well as the results of these reductions:

```
->reduce(1,2,0,0,0,3,0,3)

X0 in [  1.00 ..  2.00 ]
X1 in [  0.00 ..  0.00 ]
X2 in [  0.00 ..  3.00 ]
X3 in [  0.00 ..  1.00 ]

->reduce(0,1,0,1,0,3,0,3)

X0 in [  0.00 ..  1.00 ]
X1 in [  0.00 ..  1.00 ]
X2 in [  0.00 ..  2.00 ]
X3 in [  0.00 ..  2.00 ]

->reduce(0,3,0,3,1,1,1,1)
```

```
X0 in [  0.00 ..  2.00 ]
X1 in [  0.00 ..  2.00 ]
X2 in [  1.00 ..  1.00 ]
X3 in [  1.00 ..  1.00 ]
```

7 Related work

Solver learning is an emerging technique which was pioneered by Apt and Monfroy in [4, 5] and Abdennadher and Rigotti [1, 2].

In [4] and [5], very simple rules of the form $x_1 = s_1, \ldots, x_n = s_n \rightarrow y \neq a$ are at first considered. They provide a notion of consistency weaker than arc-consistency. Then these rules are extended by replacing equality by membership to achieve arc-consistency. The learning algorithm generates every rule to check its validity, and redundant rules are eliminated. This framework is language-independent, but the rules have a very restrictive form.

The work of [1] extends the language issue to the far more expressive framework of CHRs. The PROPMINER algorithm consists in an exploration of a user-restricted learning space for the rule's lefthand side and a computation of the righthand side, leading to the choice of the best covering rules. In [2], this work has been extended to handle prolog-like definition of constraints. This work is based on CHR and its rewriting semantics. We propose in contrast to define lower-lovel operators which yield in general faster solvers.

Other formalism compute upper bounds at run-time, such as Generalized Constraint Propagation [13], or Constructive Disjuction [16, 6], but are not designed to build statically a solver.

8 Conclusion

In this paper, we have presented a general framework of semantic approximations for finite domains CSPs. The original constraints are replaced by simpler ones and the solutions are approximated by a notion of consistency defined as the closure of a set of operators. By adding a language-based representation for both the data and the operator, we obtain a learning space in which we look for the best set of operator for the constraint's propagation. Correctness and completeness issues are adressed, in particular the notion of completeness for singletons which justify the compilation scheme of Gnu-Prolog [7].

We applied this framework to the learning of indexical operators [16, 6] which are at the core of the Gnu-Prolog system [7] and built two learning tools. The first one handles constraints of arbitrary arity and uses the simplex algorithm as optimization tool. The second is restricted to binary constraints but proposes a graphical user interface which allows the user not only to draw its constraint, but also to visualize the learned expressions and to try reductions. It uses internally a genetic algorithm to achieve the optimization.

On regular constraints, such as arithmetic or boolean constraints, Gnu-Prolog indexicals are found most of the time. On user-defined constraints, the operators found are of good quality in term of pruning power. Since our algorithm forces the correctness, the resulting solver may be used to prune the search space.

Future work includes the study of the simplification of the examples set to adress larger problems, but this may yield an incorrect solver. We are investigating both the use of safe approximations as learning examples and the use of a possibly incorrect solver, which may be faster than the correct one. In addition, more learning space may be used by relaxing the language biases.

References

1. S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In Rina Dechter, editor, *Principles and Practice of Constraint Programming*, volume 1894 of *Lecture Notes in Computer Science*, pages 18–34, Singapore, 2000. Springer.
2. S. Abdennadher and C. Rigotti. Towards inductive constraint solving. In Toby Walsh, editor, *International Conference on Principles and Practice of Constraint Programming, CP'2001*, volume 2239 of *LNCS*, pages 31–45. Springer, Nov 26 - Dec 1 2001.
3. K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 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. K. R. Apt and E. Monfroy. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 1(6):713 – 750, 2001.
6. Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27(3):185–226, 1996.
7. 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.
8. Mehmet Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and A. Herold. The CHIP System: Constraint Handling in Prolog. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, Argonne, May 1988. Springer.
9. François Fages, Julian Fowler, and Thierry Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37(1-3):185–212, 1998.
10. Thom Früwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
11. 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.
12. 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.
13. Thierry Le Provost and Mark Wallace. Generalized constraint propagation over the CLP Scheme. *Journal of Logic Programming*, 16:319–359, 1993.
14. Jean-Francois Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In John W. Lloyd, editor, *International Logic Programming Symposium*, pages 513–527, Portland, Oregon, 1995. MIT Press.
15. Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
16. P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). draft, 1991.
17. V. N. Vapnik. *The Nature of Statistical Learning Theory.* Springer, New York, 1995.
18. Neng-Fa Zhou. A high-level intermediate language and the algorithms for compiling finite-domain constraints. In Joxan Jaffar, editor, *Joint International Conference and Symposium on Logic Programming*, pages 70–84, Manchester, UK, 15-19 June 1998. MIT Press.