Constraint Logic Programming

François Fages

These notes are extracted from a course given at Ecole Polytechnique, published in French by Ellipses, Paris, 1996 [14].

François Fages
INRIA Rocquencourt
BP 105
78153 Le Chesnay Cedex
France.
http://contraintes.inria.fr/~fages
Francois.Fages@inria.fr

Contents

Chapter 1

Introduction

There exist several fundamental connections between logic and computation, that allow to design programming languages for which the problems of program specification and program validation have a precise meaning inside the logical formalism. Logic programming in a broad sense relies on the following identifications :

Programs = Theories

Computation = Proof search

The basic idea is to identify a program to a theory, and the program's execution to proof search in that theory. In this paradigm, programming is first of all a modeling task.

In the pioneering work of A. Colmerauer and R. Kowalski in the 70's, one considered only logical clauses interpreted procedurally by a principle of automated deduction [31] [42]. The programming language Prolog was an incarnation of these ideas. D. Warren showed that Prolog could be compiled very efficiently on standard machines, but Prolog suffered from two main drawbacks: on the one hand the lack of data structures other than the logical terms, hence the unmanageable necessity of axiomatizing "the domain of discourse" in the logic, on the other hand the lack of control structures, hence the loss of declarativity for obtaining executable programs.

The emergence of constraint logic programming (CLP) as defined by J. Jaffar and J.L. Lassez in the mid 80's, contributed to correct these defects in a fundamental way [26]. The discovery was that both the theory and the technology of logic programming could be generalized to arbitrary mathematical structures given with a decidable constraint language, representing "the domain of discourse". Beside the Prolog computation structure of first-order terms with equality constraints (the Herbrand's domain), one can thus consider for instance, disequality constraints over finite or infinite terms [9], real arithmetic with linear constraints [26], integer arithmetic, finite domains [48], theories of functionality, etc. One then distinguihes in the theory, the axiomatization of the structures of interest, from the modeling of the problem to be solved. Proof search then combines hybrid techniques for logical resolution, and for constraint solving in specific structures. Constraints are solved concurrently to the logical deduction process, by numerical or symbolic algorithmic means, executed with coroutines.

CLP is a concept of programming in which the problem at hand is modeled by a set of *mathematical variables* and by a set of *relations* defined by:

i) primitive constraints, e.g. $U = R * I$,

ii) predicate symbols defined by expressions of the language, e.g.

$$\forall x \forall y \; path(x,y) \iff edge(x,y) \vee \exists z(edge(x,z) \wedge path(z,y)).$$

The resulting programming style is the one of relational model-based computing. In that paradigm a model is identified to a relation defined on the interface variables $R(x,y)$. The composition of relational models is the logical conjunction of the relations,

$$R_1|R2(x,y,z) = R_1(x,y) \wedge R_2(x,z).$$

The set of solutions of a composite model is the intersection of the solutions of the composing models. This way of structuring data and programs into composable entities is somewhat similar to the one of object-oriented languages. A fundamental difference is that contrarily to the paradigm of message passing, which is directional, relational CLP programs are *reversible*: the relation on the interface variables are defined whatever are the unknowns, the computation involves *partial information structures*, the distinction between input and output occurs at execution-time, according to the nature of the arguments, that is according to the use of the model.

The choice of the language for defining new relations is crucial for the mathematical analysis of the software, as well as for its efficient compilation into machine code. The concept of constraint logic programming doesn't exclude that this language comprises programming concepts coming from concurrent, object-oriented or imperative programming. A natural choice however from the mathematical point of view is to take the *predicate calculus* as the *kernel language* for defining new relations.

By limiting ourselves to Horn clausal theories, one defines in this way a class, denoted by CLP($\mathcal{S}$), of constraint logic programming languages parametrized by the interpretation structure $\mathcal{S}$ [26]. The class of concurrent constraint languages CC($\mathcal{S}$) [43] introduces in addition some primitives for concurrency (communication, synchronisation) based on constraint entailment. CC programs introduce a form of dynamic control with data-driven computation, which can be used to program constraint solvers by a set of concurrent agents, or to program complex resolution strategies, this opens the way to a new field of applications necessiting reactive systems instead of transformational systems. CC programs gan be given a sound and complete logical semantics in the logic programming paradigm, yet with a shift to linear logic in order to model accurately concurrency in CC [15].

The success of commercial products for constraint programming, as for instance CHIP (Cosytec), Prolog III, IV (PrologIA), ILOG-Solver (ILOG), has shown the ability of this approach of computer programming to solve declaratively industrial problems of combinatorial optimization and complex system modeling. However these successes show also that the current state of the art cannot be improved without some fundamental extensions of :

i) the languages (e.g. negation, quantifiers, optimization predicates, higher-order, static typing, object-orientation...),

ii) the constraint solvers (e.g. global constraints, explicit control, quantified constraints, combination of solvers, functional domains,...),

iii) the execution models (e.g. concurrency, reactivity, parallelism, distribution,...).

In these notes we present the class of languages CLP, by studying its mathematical properties, its principles of implementation, and some examples of applications.

Chapter 2

Logical Theories

In this chapter we recall the basic results of first-order logic which are relevant to constraint programming. In order to be self-contained we present the first-order languages, the mathematical structures which give their semantics, and their related proof systems. For a more complete treatment of these subjects, see e.g. [44].

2.1 First-Order Languages

Definition 2.1 *Let S_F be a countable set of* function symbols*, denoted by $f, g, ...$, given with their* arity *α (i.e. their number of arguments).* Constants *are function symbols with arity 0. Let V be an infinite countable set of* variables *(with arity 0), denoted by $x, y....$ The set T of* first-order terms*, denoted by $M, N, ...$, is defined inductively as the least set satisfying :*

i) $V \subset T$

ii) *if* $f \in S_F, \alpha(f) = n,\ M_1, ..., M_n \in T$ *then* $f(M_1, ..., M_n) \in T$

The set of variables occurring in a term M is denoted by $V(M)$. A term M containing a variable x will be sometimes written $M[x]$.

The *size of a term*, denoted by $|M|$, is the number of occurrences of functions, constants and variables symbols in M :

i) $|x| = 1$ if $x \in V$,

ii) if $|f(M_1, ..., M_n)| = |M_1| + ... + |M_n|$.

Remark 2.2 *Zero-order languages contain no variables. Second-order languages contain second-order terms representing functions, and allow the presence of variables in place of functions inside first-order terms (second-order terms can be substituted for second-order variables). Third-order languages contain third-order terms representing functionals and allow the presence of variables in place of functionals inside second-order terms. Omega-order languages contain terms of all finite orders.*

Definition 2.3 *Let S_P be a set of* predicate symbols*, denoted by $p, q, ...$, given with their arity α. The set P_a of (first-order)* atomic propositions *is the set*

$$P_a = \{p(M_1, ..., M_n) | p \in S_P,\ \alpha(p) = n,\ M_1, ..., M_n \in T\}.$$

Definition 2.4 *Let $S_L = \{\neg, \vee, \exists\}$ be the set of logical symbols* not, or, there exists *(existencial quantifier). The set P of (first-order)* logical formula *denoted by $\phi, \psi, ...$ is defined inductively as the least set satisfying :*

i) $P_a \subset P$

ii) $\phi \in P \Rightarrow \neg\phi \in P$

iii) $\phi, \psi \in P \Rightarrow \phi \vee \psi \in P$

iv) $x \in V,\ \phi \in P \Rightarrow \exists x\phi \in P$

The other logical symbols $\{true, \supset, \wedge, \equiv\}$ are defined as abbreviations :

$$\begin{aligned}
\phi \supset \psi &= \neg\phi \vee \psi \\
true &= \phi \supset \phi \\
\phi \wedge \psi &= \neg(\phi \supset \neg\psi) \\
\phi \equiv \psi &= (\phi \supset \psi) \wedge (\psi \supset \phi)
\end{aligned}$$

the *universal quantifier*, $\forall$, is defined as an abbreviation for :

$$\forall x\phi = \neg\exists x \neg\phi$$

Quantifiers are logical symbols that define the (universal or existential) nature of a variable in a proposition. The variables of a proposition ϕ which are not *bound* by a quantifier are said to be *free* in ϕ. In a term all variable are free. The set of *free variables* of a formula ϕ, denoted by $V(\phi)$, is defined inductively by:

i) $V(x) = \{x\}$

ii) $V(f(M_1, ..., M_n)) = \bigcup_{i=1}^{n} V(M_i)$

iii) $V(p(M_1, ..., M_n)) = \bigcup_{i=1}^{n} V(M_i)$

iv) $V(\neg\phi) = V(\phi)$

v) $V(\phi \vee \psi) = V(\phi) \cup V(\psi)$

vi) $V(\forall x\phi) = V(\exists x\phi) = V(\phi) - \{x\}$

A formula ϕ is *closed* if $V(\phi) = \emptyset$.

We write $\forall(\phi)$ (resp. $\exists(\phi)$) for the closed formula $\forall x_1 ... \forall x_n \phi$ (resp. $\exists x_1 ... \exists x_n \phi$) where $\{x_1, ..., x_n\} = V(\phi)$.

Definition 2.5 *A* clause *is a disjunction of universally quantified literals,*

$$\forall(L_1 \vee ... \vee L_n),$$

where each literal L_i is either an atomic proposition, A, (called a positive literal*), or the negation of an atomic proposition, $\neg A$ (called a* negative literal*).*

A Horn clause *is a clause having at most one positive literal.*

2.2 Mathematical Structures

A *pre-interpretation* of a first-order language is a mathematical structure composed of an interpretation *domain* D, given with a semantic function $[]$, that associates to each constant $c \in S_F$ some element $[c] \in D$, and to each function symbol $f \in S_F$ with arity $n \geq 1$, some operator $[f]\ : D^n \rightarrow D$.

A *valuation* of the variables is a function $\rho : V \rightarrow D$. The valuation of the terms, denoted by $[\ ] : T \rightarrow D$, induced by a valuation ρ of the variables and a pre-interpretation $< D, [] >$ is defined (by structural induction) by :

i) $[x]_\rho = \rho(x)$ if $x \in V$,

ii) $[c]_\rho = [c]$ if $c \in S_F$ with arity 0, and $[c] \in D$ is the element assigned to c by the pre-interpretation,

iii) $[f(M_1, ..., M_n)]_\rho = [f]([M_1]_\rho, ..., [M_n]_\rho)$ if $f \in S_F$ with $n \geq 1$, $[f]$ is the operator over D assigned to f by the pre-interpretation, and $[M_i]_\rho \in D$ is the element of D assigned recursively to the subterm M_i.

An *interpretation* $I =< D, [] >$ associates in addition to each predicate symbol $p \in S_P$ with arity n, a relation $[p] : D^n \to \{0, 1\}$.

The *truth value of an atomic proposition* $p(M_1, ..., M_n)$ in an interpretation $I =< D, [] >$ and a valuation ρ is the boolean value $[p]([M_1]_\rho, ..., [M_n]_\rho)$.

The *truth value of a logical formula* ϕ in an interpretation I and a valuation ρ is determined according to the truth value of the propositions by applying the truth tables of the logical connectors, and the following rules for the quantifiers :

$\forall x\phi$ is true in I and ρ, if for every substitution of x by an arbitrary element of the domain $d \in D$, $\phi[d/x]$ is true in I and ρ.

$\exists x\phi$ is true in I if there exists an element $d \in D$ such that $\phi[d/x]$ is true in I and ρ.

Note that the truth value of a closed formula is determined solely by the interpretation and doesn't depend on the valuation.

Definition 2.6 *An interpretation I is a* model *of a closed formula ϕ if ϕ is true in I, which is denoted by $I \models \phi$.*

A closed formula ϕ' is a logical consequence *of ϕ closed, which is denoted by $\phi \models \phi'$, if every model of ϕ is a model of ϕ'.*

Definition 2.7 *A (non-closed) formula ϕ is* satisfiable *in an interpretation I if $I \models \exists(\phi)$,* valid *in I if $I \models \forall(I)$.*

A formula ϕ is satisfiable *if $\exists(\phi)$ has a model,* valid *if every interpretation is a model of $\forall(\phi)$, which is denoted by $\models \phi$.*

Proposition 2.8 *Let ϕ and ϕ' be two closed first-order formulas. $\phi \models \phi'$ if and only if $\models \phi \supset \phi'$.*

PROOF: Let us suppose $\phi \models \phi'$. For every interpretation I, if $I \models \phi$ then $I \models \phi'$ thus $I \models \phi \supset \phi'$, otherwise $I \not\models \phi$ and we have again $I \models \phi \supset \phi'$, therefore $\models \phi \supset \phi'$.

Conversely if $I \models \phi$ then as $\models \phi \supset \phi'$, we have $I \models \phi'$, thus $\phi \models \phi'$. □

Definition 2.9 *An interpretation I is a* model of a set of closed formulas *S os I is a model of each formula in S.*

We say that a set of closed formulas S is satisfiable if S has a model, valid if every interpretation is a model of S.

The logical formulas of the predicate calculus are interpreted in arbitrary structures formed with a domain, operators and relations. A formula is valid if it is true in all the interpretations on all conceivable mathematical structures. The interest in clausal forms is that it is possible for these formulas to restrict the search of a model to only one "syntactic" structure: the Herbrand's universe.

Definition 2.10 The Herbrand's universe, *denoted by $\mathcal{H}$, of a first-order language is the set of closed terms formed on the function and constant symbols $T(S_F)$. The*Herbrand's pre-interpretation *is the algebra of closed terms, whose domain is the Herbrand's universe, the symbols of constant are interpreted by these constants themselves and the symbols of function are interpreted as term constructors:*

i) $[c] = c$

ii) $[f(M_1, ..., M_n)] = f([M_1], ..., [M_n])$

The *Herbrand's base* B_H is the set of closed atomic propositions formed on S_F and S_P. A *Herbrand's interpretation* associates a truth value to every element of the Herbrand's base. We thus identify a Herbrand's interpretation to a subset of B_H, the subset of true atomic propositions.

Proposition 2.11 *Let S be a set of clauses. S is insatisfiable if and only if S has no Herbrand's model.*

PROOF: If S admits a Herbrand's model then S is satisfiable, conversely let I be an interpretation, and let I' be the Herbrand's interpretation defined by

$$I' = \{P(M_1, ..., M_n) \in B_H \mid I \models P(M_1, ..., M_n)\}.$$

If I is a model of S, then for every valuation of the variables and for every clause $C \in S$, there exists a positive literal A (resp. negative literal $\neg A$) in C such that $I \models A$ (resp. $I \not\models A$). In particular for every valuation of the variables by elements of the domain associated to terms of the Herbrand's universe, thus for every Herbrand's valuation, there exists a literal A (resp. $\neg A$) such that $I' \models A$ (resp. $I' \not\models A$). Therefore I' is a Herbrand's model of S. □

The study of the satisfiability of a set of clauses can thus be restricted to the only "syntactic" interpretations that are Herbrand's interpretations. It is worth noting that this property is flase for more general logical formulas, in particular for the existentially quantified formulas. For instance $p(a) \wedge \exists x \neg p(x)$ is satisfiable but has no Herbrand's model if a is the only constant symbol. It doesn't suffice either to consider an infinite set of constants for extending the property to formulas containing arbitrary alternate sequences of quantifiers.

It is however possible to associate to every formula ϕ a clausal formula ϕ^s, called the Skolem normal form of ϕ, which is satisfiable if and only if ϕ is satisfiable. The first transformation consists in putting the formula in *prenex conjunctive normal form*, that is under the form

$$\Lambda x_1 ... \Lambda x_k \ ((L_1^1 \vee ... \vee L_{k_1}^1) \wedge ... \wedge (L_1^n \vee ... \vee L_{k_n}^n))$$

where the L_i's are literals and each Λ is a universal or existential quantifier. This transformation needs to rename the variables which are quantified several times. The formula in prenex form is equivalent to the initial formula.

The second transformation, called Skolemisation, allows to eliminate the existential quantifiers. It consists in replacing an existentially quantified variable x by terms of the form $f(x_1, ..., x_n)$ where f is a new function symbol and the x_i's are the universally quantified variables which precede the quantification of x. The formula obtained in this way is called the *Skolem's normal form*.

Example 2.12 *For instance the Skolem's normal form of $\forall x \exists y \forall z \ p(x, y, z)$ is the formula $\forall x \forall z \ p(x, f(x), z)$ where f is a new function symbol.*

The Skolemisation preserves the satisfiability but not necessarily the validity (because the Skolemisation doesn't commute with the negation). For instance , the formula $\forall x \exists y \ p(x) \supset p(y)$ is valid, but its Skolem's normal form, $\forall x \ p(x) \supset p(f(x))$ is of course satisfiable but not valid.

Proposition 2.13 (Skolem's proposition) *Any formula ϕ is satisfiable if and only if its Skolem's normal form ϕ^s is satisfiable.*

PROOF: If $M \models \phi$ then one can choose an interpretation of the Skolem's function symbols in ϕ^s according to the M-valuation of the existential variables of ϕ such that $M \models \phi^s$. Conversely, if $M \models \phi^s$, the interpretation of the Skolem's functions in ϕ^s gives a valuation of the existential variables in ϕ which shows that $M \models \phi$. □

2.3 Proofs

In this section we study the relation of deduction, denoted by $\vdash$, which allows to build proofs of logical formulas. The fondamental problem of mathematical logic is the study of the two relations $\vdash$ and $\models$. These relations play complementary roles. In general the definition of the semantics $\models$ doesn't provide a decision procedure. It is the case in propositional logic with the method of truth tables, but this method doesn't generalize. The study of the relation of deduction then respond to this aim. Conversely, the study of the semantics of a theory defined by the relation of deduction, allows to prove that the theory is not contradictory, simply by exhibiting a model.

A *logical theory* $\mathcal{T}$ is a formal system constituted by:

i) a first-order language formed on a alphabet V, S_F, S_P, S_L,

ii) logical axioms:

   $\neg A \vee A$ (excluded middle),

   $A[x \leftarrow B] \supset \exists x\ A$ (axiom of substitution),

iii) a set of closed formulas called the non-logical axioms, and denoted by $\mathcal{T}$ (as the logical components are invariant),

iv) logical inference rules:

   $\dfrac{A}{B \vee A}$ (Weakening),

   $\dfrac{A \vee A}{A}$ (Contraction),

   $\dfrac{A \vee (B \vee C)}{(A \vee B) \vee C}$ (Associativity),

   $\dfrac{A \vee B \quad \neg A \vee C}{B \vee C}$ (Cut),

   $\dfrac{A \supset B \quad x \notin V(B)}{\exists x A \supset B}$ (Existential introduction).

We note $\mathcal{T} \vdash \phi$ the derivation of the formula ϕ in this formal system, i.e. by the application of the inference rules and of the logical and non logical axioms in $\mathcal{T}$.

A theory $\mathcal{T}$ is *contradictory* (or *inconsistent*) if $\mathcal{T} \vdash f$, *consistent* otherwise.

Theorem 2.14 (Deduction theorem) *Let $\mathcal{T}$ be a first-order logical theory. For all formulas $\phi, \psi \in P$ we have $\mathcal{T} \vdash \phi \supset \psi$ iff $\mathcal{T} \cup \{\phi\} \models \psi$.*

PROOF: In the direction of the implication ($\Rightarrow$) the result is immediat by the cut rule. Conversely the proof is by induction on the derivation of the formula ψ. □

Theorem 2.15 (Validity) *Let $\mathcal{T}$ be a first-order logical theory, and ϕ a formula. If $\mathcal{T} \vdash \phi$ then $\mathcal{T} \models \phi$.*

PROOF: By induction on the length of the deduction of ϕ. □

Corollary 2.16 *If $\mathcal{T}$ has a model then $\mathcal{T}$ is consistent*

PROOF: We show the contrapositive: if $\mathcal{T}$ is contradictory, then $\mathcal{T} \vdash f$, thus $\mathcal{T} \models f$, i.e. $\mathcal{T}$ has no model. □

Theorem 2.17 (Gödel Completeness Theorem (first form)) *A theory is consistent iff it has a model.*

PROOF: The idea is to construct a Herbrand's model of the theory supposed to be consistent, by interpreting by true the closed atoms which are theorems of $\mathcal{T}$, and by false the closed atoms whose negation is a theorem of $\mathcal{T}$. If the theory is not complete, this doesn't provide a model, we thus complete the theory by adding axioms in such a way as to obtain a complete consistent theory. For this it is necessary also to extend the alphabet in order to obtain a saturated theory, that is a theory such that if $\mathcal{T} \vdash \exists x A$ then there exists a term M of the Herbrand's universe such that $\mathcal{T} \vdash A[M/x]$. See for instance [44]. □

Theorem 2.18 (Gödel's Completeness Theorem (second form)) *Let $\mathcal{T}$ be a logical theory first-order, and ϕ be a formula,*

$$\mathcal{T} \models \phi \;\Leftrightarrow\; \mathcal{T} \vdash \phi.$$

PROOF: If $\mathcal{T} \models \phi$ then $\mathcal{T} \cup \{\neg\phi\}$ has no model, thus by the completeness theorem in first form, $\mathcal{T} \cup \{\neg\phi\} \vdash f$, hence by the deduction theorem $\mathcal{T} \vdash \neg\neg\phi$, and thus by the cut rule with the axiom of excluded middle (plus weakening and contraction) we get $\mathcal{T} \vdash \phi$. The converse is the theorem of validity. □

Gödel's completeness theorem expresses the adequation between the semantic notion of validity of a formula in all the models of the theory, and the syntactic notion of deduction. The following section shows the use of this theorem to decide the validity of a formula in a theory.

2.4 Completeness of theories and decidability of structures

Definition 2.19 *A theory $\mathcal{T}$ is* axiomatic *if the set of non logical axioms is recursive (i.e. membership to this set can be decided by an algorithm).*

For instance the theories containing a finite number of non logical axioms are trivially axiomatic. Gödel's completeness theorem shows that in an axiomatic theory, the truth in all the models of the theory is recursively enumerable. The validity of a formula can indeed be verified in finite time by searching for all possible proofs (still the satisfiable not valid formulas are not recursively enumerables). This is what shall be done in logic programming with a very simple proof system which is well suited to a machine implementation, and which is complete for the Horn clause formulas.

Definition 2.20 *A theory is* complete *if for every closed formula ϕ, either $\mathcal{T} \vdash \phi$ or $\mathcal{T} \vdash \neg\phi$.*

A structure S is axiomatizable *if there exists a complete axiomatic theory T such that S is a model of T.*

In a complete axiomatic theory, we can decide whether an arbitrary formula is satisfiable or not. This will be the expected situation for the language of constraints. The question is then to know whether there exist complete axiomatic theories for the structures of interest, and how these complete theories can be turned into efficient algorithms.

The compactness theorem of the first-order logic provides a powerful tool to study structures and theories.

Theorem 2.21 (Compactness theorem) *Let $\mathcal{T}$ be a logical first-order theory, and ϕ be a formula. $\mathcal{T} \models \phi$ iff $\mathcal{T}' \models \phi$ for some finite part $\mathcal{T}'$ of $\mathcal{T}$.*

PROOF: By the completeness theorem, $\mathcal{T} \models \phi$ iff $\mathcal{T} \vdash \phi$. As the proofs are finite, they use only a finite part of the non logical axioms of $\mathcal{T}$. Therefore $\mathcal{T} \models \phi$ iff $\mathcal{T}' \models \phi$ for some finite part $\mathcal{T}'$ of $\mathcal{T}$. □

Corollary 2.22 *A theory $\mathcal{T}$ has a model iff every finite part of $\mathcal{T}$ has a model.*

PROOF: $\mathcal{T}$ has no model iff $\mathcal{T} \models f$, iff for some finite part $\mathcal{T}'$ of $\mathcal{T}$ $\mathcal{T}' \models f$, iff some finite part of $\mathcal{T}$ has no model. □

For instance we can use this theorem to show that there doesn't exist a logical (first-order) theory of finite fields. Indeed let us suppose the opposite, let $\mathcal{T}$ be such a theory whose only models are finite fields. Let us consider the axioms A_n which state that there exist at least n distinct elements, for instance A_3 is the formula $\exists x \exists y \exists z\ x \neq y \wedge y \neq z \wedge z \neq x$. Let $\mathcal{T}'$ be the theory formed of $\mathcal{T}$ and of all the A_n's. Then by hypothesis, $\mathcal{T}'$ has no model, thus there exists a finite part $\mathcal{T}''$ of $\mathcal{T}'$ which has no model. However let n_0 be an index greater than all the n's such that $A_n \in \mathcal{T}''$, and let $\mathcal{C}$ be a finite field of more than n_0 elements, then $\mathcal{C}$ is a model of $\mathcal{T}''$, a contradiction.

The compactness theorem can also be used to construct models. This will be done in the last chapter to obtain some completeness results w.r.t. the principe of resolution for constraint logic programmming (cf. 5.19, 5.28).

Another classic use of the compactness theorem is to generalize to infinite graphs the results obtained for the finite graphs.

Solved Exercise 2.23 *In 1976 Appel and Haken proved the famous four-colors conjecture: any map can be colored with four colors (i.e. the vertices of any finite planar graph can be colored with four colors in such a way as two adjacent vertices have different colors). Extend the result to infinite planar graphs by using the compactness theorem of first-order logic.*

Solution: *Let G be an infinite planar graph. We associate to each vertex of G a symbol of constant, and we consider the first-order language formed on this infinite set of constants plus four unary predicates, c_1, c_2, c_3, c_4. Let $\mathcal{T}$ be the (possibly infinite) set of logical first-order formulas:*

i) $\forall x\ \bigvee_{i=1}^{4} c_i(x)$,

ii) $\forall x\ \bigwedge_{1 \leq i < j \leq 4} \neg(c_i(x) \wedge c_j(x))$,

iii) $\bigwedge_{i=1}^{4} \neg(c_i(a) \wedge c_i(b))$ *for every pair of constants $\{a, b\}$ which denote adjacent vertices in G.*

Clearly any coloring of G with 4 colors gives a model of $\mathcal{T}$, and conversely if $\mathcal{T}$ has a model then G can be colored with four colors as it is sufficient to choose for each vertex a the color $c_i(a)$ which is true in that model.

Let $\mathcal{T}'$ be any finite part of $\mathcal{T}$, and let G' be the (finite) subgraph of G containing the vertices which appear in $\mathcal{T}'$. As G' is finite and planar it can be colored with 4 colors, thus $\mathcal{T}'$ has a model.

Now as every finite part of $\mathcal{T}$ is satisfiable, we deduce from the compactness theorem that $\mathcal{T}$ is satisfiable. Therefore every infinite planar graph can be colored with four colors.

The structure of natural numbers, $\mathbf{N}$ with 0, s (successor), $+$ and $=$, that is the linear fragment of integer arithmetic, can be shown to be decidable. *Presburger's arithmetic* $(\mathbf{N}, 0, s, +, =)$ can be presented with a complete axiomatic theory, formed with the *standard equality axioms*:

E_1 : $\forall x\ x = x$,

E_2 : $\forall x \forall y\ x = y \to s(x) = s(y)$,

E_3 : $\forall x \forall y \forall z\ x = y \wedge z = v \to (x = z \to y = v)$,

plus the stronger equality axioms:

E_4, Π_1: $\forall x \forall y\ s(x) = s(y) \to x = y$,

E_5, Π_2: $\forall x\ 0 \neq s(x)$,

the definition of $+$:

Π_3: $\forall x\ x + 0 = x$,

Π_4: $\forall x\ x + s(y) = s(x + y)$.

and the induction principle:

Π_5: $\phi[x \leftarrow 0] \wedge (\forall x\ \phi \to \phi[x \leftarrow s(x)]) \to \forall x \phi$ for every formula ϕ.

Note that the following strong equality axioms

E_6: $\forall x\ x \neq s(x)$,

E_7: $\forall x\ x = 0 \vee \exists y\ x = s(y)$,

are provable by induction. The induction principle cannot be replaced by E_6 and E_7 but there does exist presentation of Presburger's arithmetic without the induction schema.

Peano's arithmetic contains moreover two axioms for $\times$:

Π_6: $\forall x\ x \times 0 = 0$,

Π_7: $\forall x \forall y\ x \times s(y) = x \times y + x$,

This is not sufficient however for obtaining a complete theory, and such a complete cannot exist for the integers with multiplication:

Theorem 2.24 (Gödel's incompleteness theorem) *Any consistent axiomatic extension of Peano's arithmetic is incomplete.*

PROOF: See for instance [44]. The keystone of this very beautiful proof is the liar paradox of Epimenides (600 bc) which says: "I lie", combined with Cantor's diagonal argument (cf . section 4.17). The idea of the proof is to construct in the language of Peano's arithmetic Π a formula ϕ which is true in the structure of natural numbers $\mathbf{N}$ if and only if ϕ is not provable in Π. As $\mathbf{N}$ is a model of Π, ϕ is necessarily true in $\mathbf{N}$ and not provable in Π, hence Π is incomplete. The construction of such a formula ϕ uses an arithmetization of the syntax in which every formula is associated with an integer, called its Gödel number. One then constructs a unary relation on $\mathbf{N}$ indicating whether its argument is the Gödel number of a provable formula in Π, and one exhibits a formula expressing its own negation (a similar construction of a Prolog program is given in section 5.4, for showing the indecidability of the least Herbrand's model of a logic program). This shows that Peano's arithmetic is incomplete. The construction doesn't depend so much however on the axioms of Peano than on the expressive power of the language of arithmetic, and the proof holds in fact for any consistent extension of Peano's arithmetic. $\square$

Corollary 2.25 *The structure* $(\mathcal{N}, 0, 1, +, *)$ *is not axiomatizable.*

Gödel's incompleteness theorem refutes the existence of (even infinite) complete axiomatic theories for structures of interest such as the natural numbers. Fixing the domain of discourse in constraint programming is thus not harmless, as this time, Gödel's incompleteness theorem can apply. It will be possible for some structures only, or for non axiomatizable structures by restricting the language of constraints to a decidable fragment. Of course the theoretical decidability doesn't suffice either, we will be especially interested by decidable fragments with a low algorithmic complexity, for which moreover incremental algorithms can be designed.

Chapter 3

Constraint logic programs

The basic idea of *constraint logic programmming*, introduced by J. Jaffar and J.L. Lassez, is to fix a structure of interpretation $\mathcal{S}$ representing the "domain of discourse", and to distinguish in a logic program the language of constraints on $\mathcal{S}$ supposed to be decidable, from the language of predicates defined by logical formulas. The logical formulas allowed for the definition of predicates are restricted to be Horn clauses of the form:

$$A \leftarrow c_1, ..., c_m | A_1, ...A_n$$

where the c_i are constraints and the A_j are atoms. These clauses have both a declarative logical meaning: A is true if $c_1, ..., c_m, A_1, ...A_n$ are true, and a very simple procedural interpretation: to show A it is sufficient to satisfy $c_1, ..., c_m$ and to show $A_1, ..., A_n$. In this way one defines a class of programming languages, denoted by $CLP(\mathcal{S})$, parametrized by the structure $\mathcal{S}$.

3.1 Constraints

We consider a first-order language defined by

i) a set S_F of symbols of constants and of functions,

ii) a set S_C of predicate symbols supposed to contain *true* and $=$,

iii) a countable set V of variables.

An *atomic constraint* is an atomic proposition of this language. We assume a set of *basic constraints*, supposed to be closed by variable renaming, and to contain all atomic constraints. The *language of constraints* is the closure by conjonction and existential quantification of the set of basic constraints. Constraints will be denoted by $c, d, ...$

Intuitively the basic constraints are the formulas that the constraint solver can deal with, they define the decidable fragment we are interested in, this fragment can authorize restricted forms of negation or of universal quantification, without containing necessarily all first-order formulas.

The closure by conjonction of the constraint language is essential to the principle of resolution. The closure by existential quantification has not the same status, it serves only to check the satisfiability of the projection of a computed constraint on the variables of interest (cf. 3.5).

The interpretation of constraints is supposed to be fixed by the choice of some mathematical structure $\mathcal{S} = (\mathcal{D}, E, O, R)$ formed with:

i) a domain $\mathcal{D}$,

ii) a set $E \subseteq \mathcal{D}$ of distinguished elements associated to each constant, denoted by $[c]$ for every $c \in S_F$ with arity 0,

iii) a set O of operators on $\mathcal{D}$ associated to each function symbol, denoted by $[f] : D^n \to D$ for every $f \in S_F$ with arity n,

iv) a set R of relations on D associated to each constraint predicate symbol, denoted by $[p] : D^n \to \{0, 1\}$ for every $p \in S_C$ with arity n.

An $\mathcal{S}$-valuation is a function $\rho : V \to \mathcal{D}$ that extends to terms by morphism. If $\mathcal{S} \models c\rho$ we say that c is *satisfiable* and that ρ is a *solution* of c, otherwise we have $\mathcal{S} \models \neg c\rho$.

We shall assume that in the structure $\mathcal{S}$, *the constraint satisfiability problem is decidable.* We shall thus suppose without loss of generality that $\mathcal{S}$ is presented by an axiomatic theory $\mathcal{T}$ defined on the alphabet S_C, S_F, satisfying:

1. (soundness) $\mathcal{S} \models \mathcal{T}$
2. (completeness for constraint satisfaction) for every constraint c, either $\mathcal{T} \vdash \exists(c)$, or $\mathcal{T} \vdash \neg\exists(c)$.

Under these assumptions we have that $\mathcal{S} \models \exists(c)$ iff $\mathcal{T} \vdash \exists(c)$. We do not demand however that $\mathcal{T}$ is a complete theory because we are merely interested by the existential conjunctive fragment of the language of constraints. If the constraints can be arbitrary first-order formulas, then condition 2) does express that $\mathcal{T}$ is a complete theory.

3.2 CLP($\mathcal{S}$) Programs

We consider also a set of predicate symbols S_P disjoint from S_C, representing relations defined by program. In the following we call *atom* an atomic proposition formed on S_P, S_F and V exclusively.

Definition 3.1 *A* constraint logic program clause *is a clause with exactly one positive literal* $\forall(A \vee \neg c_1 \vee ... \neg c_n \vee \neg A_1 \vee ... \vee \neg A_n)$ *where* $m \geq 0$, $n \geq 0$, *the* c_i*'s are atomic constraints and the* A_j*'s are atoms. A clause of program is denoted by*

$$A \leftarrow c_1, ..., c_m | A_1, ...A_n$$

or

$$A \leftarrow c|\alpha$$

where $c = c_1 \wedge ... \wedge c_m$, *and where* α *denotes the sequence of atoms* $A_1, ..., A_n$. A *is called the* head *of the clause, and* $c|\alpha$ *the* body*. The* local variables *of the clause are the variables which appear uniquely in the body of the clause.*

A constraint logic program *is a finite set of program clauses.*

Definition 3.2 *A* goal clause *is a clause without positive literal*

$$\forall(\neg c_1 \vee ... \neg c_n \vee \neg A_1 \vee ... \vee \neg A_n)$$

A goal*, denoted by*

$$c_1, ..., c_k | A_1, ..., A_n$$

or according to the previous notations by

$$c|\alpha$$

stands for the formula $c_1 \wedge ... \wedge c_k \wedge A_1 \wedge ... \wedge A_n$.

The reason for differentiating the logical formula associated to a goal from the one associated to a goal clause is that from the point of view of theorem proving, a refutation expresses that the set of program clauses P with the *goal clause* G, $P \cup G$, is unsatisfiable, whereas from the point of view of programming, a successful derivation expresses that the *goal* G is satisfiable, $P \models \exists(G)$, both viewpoints are obviously equivalent as $P \models \exists(G)$ if and only if $P \cup \neg\exists(G)$ is unsatisfiable, and the negation of the logical formula $\exists G$ associated to a goal G is indeed a goal clause $\neg\exists G$. In the following we shall be mainly concerned with the programming language point of view, hence we shall manipulate goals, rather than goal clauses.

In order to simplify the proofs, we shall consider programs and goals in *normal form*, in which the atoms contain no function symbol. There is obviously no loss of generality as every program or goal can be transformed under this form by introducing new variables and equality constraints between these variables and the terms inside the atoms. For instance the normal form of the clause $p(x+1) \leftarrow p(x-1)$ is $p(y) \leftarrow y = x+1 \wedge z = x-1 \mid p(z)$.

The CLP programs are parametrized by the structure $\mathcal{S}$ which fixes the interpretation of the constraint language. An $\mathcal{S}$*-interpretation* of the language augmented with predicate symbols in S_P associates in addition to every $p \in S_P$ with arity n, a relation $[p] : \mathcal{D}^n \rightarrow \{0,1\}$. An $\mathcal{S}$*-model* of a program P is an $\mathcal{S}$-interpretation model of P. The $\mathcal{S}$*-base*, denoted by $B_{\mathcal{S}}$, is the set of atoms valued in $\mathcal{S}$:

$B_{\mathcal{S}} = \{p(x_1, ..., x_n)\rho \mid p \in S_P$ of arity n and ρ is an $\mathcal{S}$-valuation $\}$.

An $\mathcal{S}$-interpretation can thus be identified to a subset of $B_{\mathcal{S}}$ formed with the atoms which are true in the interpretation. Clearly $B_{\mathcal{S}}$ is a model of every constraint logic program on $\mathcal{S}$. In the following (cf. 5.4) we shall show the existence of a least $\mathcal{S}$-model, denoted by $M_P^{\mathcal{S}}$.

The logical meaning of a CLP program allows to define several declarative semantics according to the observation we are interested in, for instance:

- only the satisfiability of a goal, $\exists(G)$, (i.e. theorem proving point of view),
- or the constraints which imply a goal, $c \supset G$, (i.e. programming language point of view),

Furthermore we can consider:

i) the logical consequences of the program and of the theory of the structure (proper logical semantics),

$$(1)\ \ P, \mathcal{T} \models \exists(G) \qquad (4)\ \ P, \mathcal{T} \models c \supset G,$$

ii) the logical consequences of the program in all the $\mathcal{S}$-models of the program (logical semantics with a fixed pre-interpretation),

$$(2)\ \ P \models_{\mathcal{S}} \exists(G) \qquad (5)\ \ P \models_{\mathcal{S}} c \supset G,$$

iii) the truth in the least $\mathcal{S}$-model of the program (algebraic semantics),

$$(3)\ \ M_P^{\mathcal{S}} \models \exists(G) \qquad (6)\ \ M_P^{\mathcal{S}} \models c \supset G.$$

In the following we shall show the equivalences (1) $\Leftrightarrow$ (2) $\Leftrightarrow$ (3) and (4) $\Rightarrow$ (5) $\Leftrightarrow$ (6). Only (4) leads to a notion of correct answer weaker than (5) and (6). In a first approximation we shall retain the notion of correct answer to a goal given by the logical semantics in a fixed structure $\mathcal{S}$ (5).

Definition 3.3 *Let P be a CLP($\mathcal{S}$) program. Let G be a goal. A constraint c is a* semi-correct answer *if:*

$$P \models_{\mathcal{S}} \forall(c \supset G)$$

c is a correct answer *if furthermore c is $\mathcal{S}$-satisfiable:*

$$P \models_{\mathcal{S}} \exists(c)$$

The notion of semi-correct answer is introduced to modelize the case where the constraint solver effectively used in a CLP($\mathcal{S}$) system is not complete (e.g. 4.18, 4.6). The satisfiability of computed constraints is partially checked in such systems, the computed answers are thus semi-correct answers in general. For sake of simplicity however, the principle of resolution is defined in the next section with a complete check of satisfiability.

3.3 Procedural interpretation

The practical justification for restricting the definition of predicates to Horn clause formulas, is that it is possible to associate to such formulas an extremely simple proof system, reduced to a single inference rule, called CSLD resolution. We present this inference rule by a rewriting relation on goals.

Definition 3.4 *Let P be a constraint logic program on $\mathcal{S}$. The rewriting relation $\longrightarrow$ on goals is defined as the least relation satisfying the following principle of CSLD resolution*[1]*:*

$$\frac{(p(N_1,...,N_k) \leftarrow c'|A_1,...,A_n)\theta \in P \qquad \mathcal{S} \models \exists(c \wedge M_1 = N_1 \wedge ... \wedge M_k = N_k \wedge c')}{(c|\alpha, p(M_1,...,M_k), \alpha') \longrightarrow (c, M_1 = N_1, ..., M_n = N_n, c' \mid \alpha, A_1, ..., A_n, \alpha')}$$

where θ is a renaming substitution of the program clause with new variables.

The atom $p(M_1,...,M_n)$ in the goal to reduce is called the *selected atom*. Note that there is no rewriting if the resulting constraint is not $\mathcal{S}$-satisfiable. We write $G \longrightarrow_C G'$ for a step of resolution with the clause $C \in P$, and we note $\longrightarrow^*$ the reflexive transitive closure of $\longrightarrow$.

A *CSLD derivation* for a goal G is a finite or infinite sequence of goals $(G_j)_{j\geq 0}$, and of variants of program clauses C_j, such that $G_0 = G$ and $G_j \longrightarrow_{C_j} G_{j+1}$ for every $j \geq 0$.

A *successful derivation* (or *CSLD refutation*) is a finite CSLD derivation which terminates with a goal containing constraints only.

Definition 3.5 *Let P be a program CLP($\mathcal{S}$). A* computed answer *for a goal G is a constraint c obtained by a CSLD refutation from G:*

$$G \longrightarrow^* c|\Box$$

The projected computed answer *is the constraint $\exists x_1...\exists x_k\ c$ where $\{x_1,...,x_k\} = V(c) \setminus V(G)$.*

Example 3.6 *Consider the following CLP($\mathbf{N}$) program:*
$p(0) \leftarrow$
$p(x+1) \leftarrow p(x)$
The goal $p(y)$ has the following successful derivations:
$p(y) \longrightarrow y = 0|\Box$
$p(y) \longrightarrow y = y_1 + 1|p(y_1) \longrightarrow y = y_1 + 1 \wedge y_1 = 0|\Box$
etc.
The projected computed constraints are $y = 0$, $y = 1$, etc.

[1] CSLD stands for Linear resolution for Definite programs with Constraints and Selected atom.

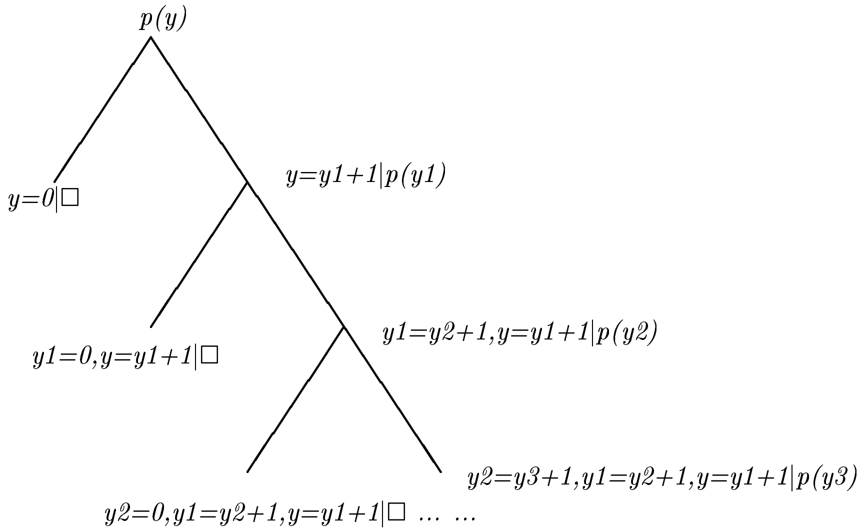

Figure 3.1: Infinite CSLD tree of the example 3.6.

Lemma 3.7 ($\wedge$-compositionality) *c is a computed answer for the goal $(d|A_1, ..., A_n)$, if and only if there exist computed answers $c_1, ..., c_n$ for the goals $true|A_1, ..., true|A_n$, such that $c = d \wedge \bigwedge_{i=1}^{n} c_i$ is satisfiable.*

PROOF: By induction on the length of the derivation. □

Corollary 3.8 *Independance of the selection strategy Let R be a selection strategy for the atom to select at each resolution step. If c is a computed answer with the strategy R for the goal G, then for every strategy R', there exists a computed answer c' with the strategy R' for the goal G such that $\mathcal{S} \models c \leftrightarrow c'$.*

The independance of the selection strategy shows that for the observation of successes, it is possible to restrict the search for derivations from a goal G by fixing an arbitrary selection strategy.

Definition 3.9 *A* CSLD derivation tree *for a goal G is the tree of all CSLD derivations obtained from G by fixing a selected atom in each node.*

To enumerate all the succeses to a goal G, the independance of the selection strategy thus shows that it is sufficient to search in an arbitrary CSLD derivation tree for G.

Chapter 4

Examples

4.1 CLP($\mathcal{H}$) and Prolog

In the class of programming languages CLP($\mathcal{H}$), the interpretation structure is the algebra of first-order terms, the Herbrand's domain $\mathcal{H}$.

The programming language *Prolog* is an implementation of CLP($\mathcal{H}$) in which:

i) the constraints are only equalities between terms, they are solved by a unification algorithm (some implementations of Prolog treat also disequality constraints by a mechanism of coroutines, cf. predicate `dif(X,Y)`),

ii) the selection strategy consists in solving the atoms from left to right according to their order in the goal, the atoms to solve are thus implemented with a stack (some implementations have a mechanism of coroutines which modifies the selection strategy by delaying the selection of some atoms as long as a variable is not instanciated, e.g. predicate `freeze(X,G)`),

iii) the search strategy consists in searching the derivation tree *depth-first* by *backtracking*.

In Prolog the syntax of the program clauses is
`A :- B1,...,Bn.`,
`A.`
the syntax of the goals is
`?- A1,...,Year.`.
The interpreter enumerates the computed answers to a goal by typing `;` after the prompt.

Program 4.1 *The deductive data bases give a first example of Prolog programs on an alphabet of constants without function symbols:*

```
gdfather(X,Y):-father(X,Z),parent(Z,Y).
gdmother(X,Y):-mother(X,Z),parent(Z,Y).

parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).

father(alphonse,chantal).
mother(emilie,chantal).
mother(chantal,julien).
```

```
father(julien,simon).

| ?- gdfather(X,Y).

X = alphonse, Y = julien ? ;

no

| ?- gdmother(X,Y).

X = emilie, Y = julien ? ;

X = chantal, Y = simon ? ;

no
```

Program 4.2 *The introduction of a binary function symbol allows to represent the list structure, the usual relations on lists can be defined by simple programs:*

```
member(X,cons(X,L)).
member(X,cons(Y,L)):-member(X,L).

append(nil,L,L).
append(cons(X,L),M,cons(X,N)):-append(L,M,N).

| ?-  member(X,cons(a,cons(b,cons(c,nil)))).

X = a ? ;

X = b ? ;

X = c ? ;

no

| ?- member(X,Y).

Y = cons(X,_A) ? ;

Y = cons(_B,cons(X,_A)) ? ;

Y = cons(_C,cons(_B,cons(X,_A))) ? ;

Y = cons(_D,cons(_C,cons(_B,cons(X,_A)))) ? ;

Y = cons(_E,cons(_D,cons(_C,cons(_B,cons(X,_A))))) ?

yes

| ?- append(cons(a,cons(b,nil)),cons(c,cons(d,nil)),L).

L = cons(a,cons(b,cons(c,cons(d,nil)))) ? ;

no
```

Program 4.3 *Lists have a special syntax in Prolog:* `[X|L]` *stands for* `cons(X,L)` *and* `[]` *for* `nil`*. The naive program for reversing a list has a quadratic time complexity, a standard technique for obtaining a reverse program of linear time complexity is to use a third argument as an accumulator.*

```
append([],L,L).
append([X|L],L2,[X|L3]):-append(L,L2,L3).

reverse([],[]).
reverse([X|L],R):-reverse(L,K),append(K,[X],R).

| ?- reverse([a,b,c,d],M).
M = [d,c,b,a] ? ;
no

| ?- reverse(M,[a,b,c,d]).
M = [d,c,b,a] ?


rev(L,R):-rev_lin(L,[],R).

rev_lin([],R,R).
rev_lin([X|L],K,R):-rev_lin(L,[X|K],R).

| ?- reverse(X,Y).
X = [], Y = [] ? ;
X = [_A], Y = [_A] ? ;
...
```

Program 4.4 *The implementation of the various algorithms for sorting is straightforward, predefined predicates can be used for comparing integers.*

```
quicksort([],[]).
quicksort([X|L],R):-
    partition(L,Linf,X,Lsup),
    quicksort(Linf,L1),
    quicksort(Lsup,L2),
    append(L1,[X|L2],R).

partition([],[],_,[]).
partition([Y|L],[Y|Linf],X,Lsup):-
    Y=<X,
    partition(L,Linf,X,Lsup).
partition([Y|L],Linf,X,[Y|Lsup]):-
    Y>X,
    partition(L,Linf,X,Lsup).
```

Program 4.5 *A (non-deterministic) context-free grammar can be directly translated in a Prolog program. The first Prolog interpreter was designed in 1972 by A. Colmerauer for this purpose. For example the grammar:*

sentence :: nounphrase, verbphrase;
nounphrase :: determiner, noun | noun;
verbphrase :: verb — verb, nounphrase;
verb :: [eats];
determiner :: [the];

noun :: [monkey] | [banana];
can be systematically translated in the following Prolog program for parsing and synthesis:

```
sentence(L):-nounphrase(L1), verbphrase(L2), append(L1,L2,L).

nounphrase(L):- determiner(L1), noun(L2), append(L1,L2,L).
nounphrase(L):- noun(L).

verbphrase(L):- verb(L).
verbphrase(L):- verb(L1), nounphrase(L2), append(L1,L2,L).

verb([eats]).

determiner([the]).

noun([monkey]).
noun([banana]).

| ?- sentence([the,monkey,eats]).

yes

| ?- sentence([the,eats]).

no

| ?- sentence(L).

L = [the,monkey,eats] ? ;

L = [the,monkey,eats,the,monkey] ? ;

L = [the,monkey,eats,the,banana] ? ;

L = [the,monkey,eats,monkey] ?

yes
```

The basic operation of a Prolog interpreter is thus the solving of equality constraints over first-order terms, with an unbounded signature, i.e. a signature containing an infinite set of function symbols for each arity. The equality in $\mathcal{H}$ can be completely axiomatized by adding few axioms to the standard equality axioms. From such a complete axiomatization one can derive a simple unification algorithm for solving equality constraints between terms.

Definition 4.6 *The Clark's equational theory CET [8] is the theory formed with the standard axioms for equality:*

E_1: $\forall x\ x = x$,

E_2 $\forall x_1, ..., x_n, y_1, ..., y_n\ x_1 = y_1 \wedge ... \wedge x_n = y_n \rightarrow f(x_1, ..., x_n) = f(y_1, ..., y_n)$ *for every* n *and every function symbol* $f \in S_F$ *with arity* n,

E_3 $\forall x_1, ..., x_n, y_1, ..., y_n\ x_1 = y_1 \wedge ... \wedge x_n = y_n \rightarrow p(x_1, ..., x_n) \rightarrow p(y_1, ..., y_n)$ *for every* n *and every predicate symbol* $p \in S_P$ *with arity* n.

plus the axioms:

E_4: $\forall x_1, ..., x_n, y_1, ..., y_n\ f(x_1, ..., x_n) = f(y_1, ..., y_n) \rightarrow x_1 = y_1 \wedge ... \wedge x_n = y_n$ *for every function symbol* $f \in S_F$ *with arity* n,

E_5: $\forall x_1, ..., x_m, y_1, ..., y_n\ f(x_1, ..., x_m) \neq g(y_1, ..., y_n)$ *for different function symbols* $f, g \in S_F$ *with arity* m *and* n *respectively,*

E_6: $\forall x\ M[x] \neq x$ *for every term* M *strictly containing* x.

One can notice that Presburger's arithmetic contains the axioms $E_1 - E_6$ for 0 and s, while E_7 simply disappears here as we have an infinite set of function symbols.

Proposition 4.7 *$\mathcal{H}$ is a model of CET.*

Exercise 4.8 *Give a model of E_1, E_2, E_3, E_4, E_5 not satisfying E_6 (hint: imagine a structure of infinite terms).*

Give a non standard model of CET, i.e. a model of CET not isomorphic to $\mathcal{H}$ (hint: restrict the structure of infinite terms to those terms which satisfy E_6).

The theory CET is an axiomatic theory which is complete for the satisfaction of equality constraints between terms in $\mathcal{H}$. This can be shown simply by orientating the axioms of CET so as to derive an algorithm for solving equality constraints. The algorithm we obtain in this way was proposed by Herbrand in his thesis in 1930 [22], and was later rediscovered by Robinson in his seminal work on automated deduction [42].

Definition 4.9 *A* system of equations Γ *is either the symbol false* $\perp$, *or a conjonction of equations between terms* $M_1 = N_1 \wedge ... \wedge M_n = N_n$ *(true if* $n = 0$*).*

A system of equations is in solved form *if it is of the form*

$$x_1 = M_1 \wedge ... \wedge x_n = M_n$$

with $n \geq 0$ *and* $\{x_1, ..., x_n\} \cap (V(M_1) \cup ... \cup V(M_n)) = \emptyset$.

Clearly if Γ is a solved form then $CET \models \exists(\Gamma)$. The Herbrand's unification algorithm decides the satisfiability of a system Γ by computing a solved form.

Definition 4.10 *The* unification algorithm of Herbrand *simplifies a system of equations by applying the following rules:*

Dec : $f(M_1, ..., M_n) = f(N_1, ..., N_n) \wedge \Gamma \longrightarrow M_1 = N_1 \wedge ... \wedge M_n = N_n \wedge \Gamma$,
Dec$\perp$: $f(M_1, ..., M_n) = g(N_1, ..., N_m) \wedge \Gamma \longrightarrow \perp$ *if* $f \neq g$,
Triv : $x = x \wedge \Gamma \longrightarrow \Gamma$,
Var : $x = M \wedge \Gamma \longrightarrow x = M \wedge \Gamma\sigma$ *if* $x \notin V(M)$, $x \in V(\Gamma)$, $\sigma = \{x \leftarrow M\}$,
Var$\perp$: $x = M \wedge \Gamma \longrightarrow \perp$ *if* $x \in V(M)$ *and* $x \neq M$.

Lemma 4.11 (Validity) *If* $\Gamma \longrightarrow^* \Gamma'$ *then* $CET \models \Gamma \leftrightarrow \Gamma'$.

Lemma 4.12 (Termination) *There are no infinite sequence of simplifications.*

Proposition 4.13 (Decidability of unification) $CET \models \exists(\Gamma)$ *iff the irreducible form of* Γ *is a solved form.*

Corollary 4.14 (Completeness of CET) *For any equation system* Γ, *either* $CET \vdash \exists(\Gamma)$, *or* $CET \vdash \neg\exists(\Gamma)$.

Corollary 4.15 $\mathcal{H} \vdash \exists(\Gamma)$ *iff* $CET \vdash \exists(\Gamma)$.

Robinson's unification algorithm represents the unsolved part of the system as a stack and traverses the terms depth-first in left-right order. It computes furthermore, if the terms are unifiable, a substitution σ which represents the solution set. Although there exist other unification algorithms with better (linear) theoretical complexity, the Herbrand-Robinson's unification algorithm has a good practical efficiency. It is used in the implementation of Prolog, in particular in the Warren's abstract machine, with the optional ommission of the occur check (rule $Var\bot$) for efficiency reasons.

Remark 4.16 *The decidability of unification and the validity lemma show that the theory CET is complete for the existential conjunctive fragment of equality constraints. If we enrich the language of constraints by authorizing for instance disequality constraints ($\forall Y X \neq f(Y)$), or arbitrary first-order formulas, then the situation depends on the alphabet.*

If the alphabet contains an infinite set of constant symbols and function symbols, then CET is a complete theory [33] [38], the structure $\mathcal{H}$ is thus decidable.

If the alphabet is finite, formed of function symbols $f_1, ..., f_n$ with arity $n_1, ..., n_k$, then it is necessary to consider the theory CET augmented with the domain-closure axiom (DCA):

DCA: $\forall x \exists y_1 ... \exists y_n x = f_1(y_1, ..., y_{n_1}) \vee ... \vee x = f(y_1, ..., y_{n_k})$

In the case of a finite alphabet the theory CET+DCA is a complete theory [38].

Therefore in all cases the structure $\mathcal{H}$ is decidable. The class CLP($\mathcal{H}$) can thus be defined with more or less powerful constraint languages.

4.2 CLP($\mathcal{RT}$)

The absence of occur check in Prolog is not justified uniquely by (historical) reasons of practical efficiency but also by the need of programming with circular data structure, for representing cross-references for instance.

We can thus consider as computation domain the algebra of finite and infinite terms [11] or more precisely the algebra $\mathcal{RT}$ of *rational terms*, which are finite or infinite terms having a finite number of distinct subterms, and which can thus be represented by finite graphs.

If we replace in the theory CET the axiom of occur check (E_6) by a new axiom stating the existence of solutions to equation of the form $x = f(x)$, we obtain a complete theory of both the algebra $\mathcal{RT}$ and the algebra of finite and infinite terms [38], these structures are thus elementarily equivalent.

The unification algorithm of Huet [25] is a complete unification algorithm in $\mathcal{RT}$. The language Prolog II introduced by A. Colmerauer in 1982 included that unification algorithm together with a treatment of disequality constraints by a mechanism of coroutines. Today we can see Prolog II as an instance of CLP($\mathcal{RT}$). Historically, it is the theoretical study of Prolog II which lead J. Jaffar and J.L. Lassez in 1986 to the general concept of the class CLP.

4.3 CLP($\mathcal{H}/\mathcal{E}$)

By still considering term algebras, we can define the class CLP($\mathcal{H}/\mathcal{E}$) presented by an *equational theory* $\mathcal{E}$, that is a theory formed with a recursive set of identities between terms. Birkhoff's theorem shows the completeness of equational reasoning for semi-deciding equality in $\mathcal{E}$: $\mathcal{E} \models M = N$ iff $M =_{\mathcal{E}} N$, i.e. iff M and N are congruent modulo $\mathcal{E}$. It is also possible to semi-decide the satisfiability of equality constraints, $\mathcal{E} \models \exists(M = N)$, In general however the problem of $\mathcal{E}$-equality in an equational theory is undecidable.

The notion of unification in $\mathcal{H}$ can be generalized to a notion of unification with complete sets of unifiers in $\mathcal{H}/\mathcal{E}$. However these sets can be infinite, for instance the equation $f(x,a) = f(a,x)$ where f is an associative operator has an infinite base of unifiers, $\sigma_0 = \{x \leftarrow a\}, \sigma_1 = \{x \leftarrow f(a,a)\}, \sigma_2 = \{x \leftarrow f(a, f(a,a))\}, \ldots$. It can also be the case that there doesn't exist bases of unifiers in some equational theories admitting decreasing chains of more and more general unifiers.

The equational theories of interest are those in which the satisfiability of equality constraints is decidable. It is the case for instance in theories containing an associative function symbol and constants, in theories on an arbitrary alphabet with associative-commutative function symbols (in these theories there exists furthermore an a unification algorithm which computes a finite base of unifiers), in some disjoint unions of equational theories, etc. See [29] for a survey.

The equational unification algorithms, when they exist, do not always provide efficient algorithms for solving equality constraints. For instance associative-commutative unifiability is an NP-complete problem whereas the computation of a base of associative-commutative unifers is complete for the double exponential complexity class [30].

4.4 CLP(λ)

In CLP(λ) we consider the terms of the simply typed λ-calcul, they are defined by the following grammar of types t and typed expressions $e : t$:

$t \ ::= \ v \mid t_1 \to t_2$

$e : t \ ::= \ x : t \mid (\lambda x : t_1 . e : t_2) : t_1 \to t_2 \mid (e_1 : t_1 \to t_2 (e_2 : t_1)) : t_2$

The symbol λ represents the operation of formation of a function by abstraction of a variable in an expression. The other operation is the application of a function to an expression of the right type. The theory of functionality is defined by two axioms for variable renaming α and application β:

$\lambda x.e_1 =_\alpha \lambda y.e_1[y/x]$ if $y \notin V(e_1)$,

$(\lambda x.e_1)e_2 \ \to_\beta \ e_1[e_2/x]$

The type system insures the termination of β-reductions modulo α-conversion. The property of termination combined with the property of confluence of the λ-calculus, allows us to decide equality in this theory by simple rewriting:

$$e_1 \ =_{\alpha,\beta} \ e_2 \text{ iff } \downarrow_\beta e_1 \ =_\alpha \downarrow_\beta e_2.$$

However rewriting doesn't suffice to decide the satisfiability of equality constraints. For instance to solve the equation $FX = GY$ where F and G are functional variables, we can impose $F = G$, $X = Y$, or $F = \lambda x.GY$, or again $F = \lambda x.H$, $G = \lambda y.I$ with $HX = IY$ which leads back to the previous problem. Unification in higher-order languages is an undecidable problem [24], already at order 2 [19]. It is worth noting however that as the equality of typed λ-expressions is decidable, the set of unifiers of two typed λ-expressions is recursively enumerable.

Such a generalization of Prolog to higher-order logic has an extraordinary (excessive!) expressive power. As an illustration of this phenomenon, Cantor's theorem can be shown in two steps of SLD resolution where the computed substitution represents Cantor's diagonal argument!

Theorem 4.17 (Cantor's Theorem) $\mathbf{N}^{\mathbf{N}}$ *is not countable.*

PROOF: (adapted from [24]).

Let us suppose the opposite $\exists h : \mathbf{N} \to (\mathbf{N} \to \mathbf{N})\ \forall f : \mathbf{N} \to \mathbf{N}\ \exists n : \mathbf{N} \quad h(n) = f$ After Skolemisation, the formula to refute becomes $\forall F\ h(n(F)) = F$ which is equivalent to the goal clause $\forall F\ \neg h(n(F)) \neq F$.

A refutation of the goal $h(n(F) \neq F$ can be obtained by two steps of CSLD resolution with two simple properties of the natural numbers expressed by the following program:
$F \neq G \ \leftarrow \ F(N) \neq G(N).$
$N \neq s(N).$
The first program clause is used to form the first resolvant:

$$h(n\ F) \neq F \ \longrightarrow^{\sigma_1} \ (h(n\ F))(I) \neq F(I)$$

The second clause gives the refutation

$$(h(n\ F))(I) \neq F(I) \ \longrightarrow^{\sigma_2} \ []$$

with the sequential substitution

$$\sigma_2 = \{J \leftarrow h\ I\ I\}.\{I = n(F)\}.\{F = \lambda i.s(h\ i\ i)\}$$

One verifies that $(h(n\ F))(I)\sigma_2 = J\sigma_2$ and $F(I)\sigma_2 = s(J)\sigma_2$. The unifier σ_2 of the last step of resolution, which generates the contradiction, contains "the diagonal argument" of Cantor: we consider the diagonal (Hii) of H which indicates the value taken by the function number i at value i, and we construct the function f which associates to i the successor of (Hii); then we consider the value of f at $n = Nf$, that is at the number of f; the contradiction comes from the fact that on the one hand, by definition of H, $fn = (Hnn)$, and on the other hand, by construction of f, $fn = S(Hnn)$. Hence such a function h cannot exist. $\square$

It is worth noting that the unification algorithm on first-order terms could accept variables in position of function. However the unification of these expressions would be done in the first-order model of terms, and not in a theory of functionality. This generalization of first-order unification is thus not sufficient for finding the substitution of f in the previous example, but it allows to find the simple cases of higher-order unification, as in the first step of resolution.

In fact the implemented CLP(λ) systems such as λ-Prolog [39] consider *weak theories of functionality* which correspond to simple cases of higher-order unification. The interesting features of these programming languages lies in particular in their *type system* inherited from the λ-calculus, and in the natural generalisation in this context of Horn clauses to *imbricated implications*, which is the basis of an original system of modules and of powerful methods for meta-programming.

4.5 CLP($\mathcal{R}$)

The decidability of real arithmetic $(R, 0, 1, +, *, =, <)$ was shown by Tarski by showing the completeness of the axiomatic theory of real closed fields:

C_1: $(x + y) + z = x + (y + z)$,

C_2: $x + 0 = x$,

C_3: $x + (-1 * x) = 0$,

C_4: $x + y = y + x$,

C_5: $(x * y) * z = x * (y * z)$,

C_6: $x * 1 = x$,

C_7: $x \neq 0 \rightarrow \exists y\ x * y = 1$,

C_8: $x * y = y * x$,

C_9: $x * (y + z) = (x * y) + (x * z)$,

C_{10}: $0 \neq 1$,

O_1: $\neg(x < x)$,

O_2: $x < y \rightarrow (y < z \rightarrow x < z)$,

O_3: $x < y \vee x = y \vee y < x$,

O_4: $x < y \rightarrow x + z < y + z$,

O_5: $0 < x \rightarrow (0 < y \rightarrow 0 < x * y)$,

R_1: $0 < x \rightarrow \exists y\ y * y = x$,

R_2: $y_n \neq 0 \rightarrow \exists x\ y_n * x^n + y_{n-1} * x^{n-1} + ... + y_0 = 0$ for every odd integer n.

The result of completeness of this theory shows the decidability of elementary geometry, the proof is based on a method for quantifier elimination [44]. In principle this method allows us to decide the satisfiability of arbitrary first-order logical formulas on the reals, with however a tower of exponentials as algorithmic complexity... CLP(**R**) systems with that degree of generality have been realized, see for instance [23]. These prototype systems compute answers with of course widely unstable and unpredictible performances.

If we limit the constraint language to the linear existential fragment, the satisfaction problem becomes polynomial and the algorithms of linear programming provide powerful decision methods [7]. The Simplex algorithm, for example, has a quasi-linear practical complexity in the number of variables. This algorithm can moreover handle the incremental addition and deletion of constraints. For these reasons the Simplex algorithm is still the algorithm of choice for solving linear constraints in CLP(**R**) systems, while non-linear constraints are simply delayed until they become linear (e.g. with the `freeze` predicate). Several CLP(**R**) systems have been implemented since the mid 80's [26] [41], and have been succesfully used in a wide variety of applications ranging from decision support in financial domains, verification and synthesis of analogical circuits, combinatorial optimization, etc. [28].

The following CLP(**R**) program expresses the formula for computing mortgage. In the predicate $mortgage(P, T, I, B, M)$, P is the total amount, T the duration in months, I the monthly rate, B the balance, and M the monthly reimbursement. The program computes instanciated answers for different combinations of the inputs. It computes also linear constraints as answers. The last query of the example shows a case where the answer is a non-linear constraint, the satisfiability of this constraint is not checked by the system (semi-correct answer in general, cf. 3.3).

Program 4.18 *[27] Example of a CLP(**R**) program for computing mortgage.*

```
mortgage(P,T,I,B,M):- T > 0, T <= 1, B + M = P * (1 + I).
mortgage(P,T,I,B,M):- T > 1, mortgage(P * (1 + I) - M, T - 1, I, B, M).

| ?- mortgage(120000,120,0.01,0,M).

M = 1721.651381 ?

yes
```

```
| ?- mortgage(P,120,0.01,0,1721.651381).

P = 120000 ?

yes

| ?- mortgage(P,120,0.01,B,M).

P = 0.302995*B + 69.700522*M ?

yes

| ?- mortgage(999, 3, Int, 0, 400).

400 = (-400 + (599 + 999*Int) * (1 + Int)) * (1 + Int) ?

yes
```

The need for computing with complex data structure obviously remains in CLP(**R**). The structure of interest is thus not exactly **R** but more precisely the algebra $\mathcal{H}(\mathbf{R})$ of first-order terms formed on an alphabet of constant and function symbols, possibly containing arithmetic expressions in their leaves. It has been shown that under some general conditions the completeness of a theory for a structure $\mathcal{S}$ remains for the structure $\mathcal{H}(\mathcal{S})$ [46]. The following example illustrates the use of lists in CLP(**R**).

Program 4.19 *[26] Example of a CLP(**R**) program for computing the temperature on a discrete surface, or more generally for solving the Dirichlet problem for Laplace's equation by the finite difference method. The program specifies that the temperature in each interior point is the mean of its four neighbors. If the data are sufficiently instanciated, for instance the temperature on the edges is known, the answers are numerical values, otherwise they are linear constraints.*

```
laplace([H1,H2,H3|T]):-
  laplace_vec(H1,H2,H3),  laplace([H2,H3|T]). laplace([_,_]).

laplace_vec([TL,T,TR|T1],[ML,M,MR|T2],[BL,B,BR|T3]):-
  B + T + ML + MR - 4 * M = 0,
  laplace_vec([T,TR|T1],[M,MR|T2],[B,BR|T3]).
laplace_vec([_,_],[_,_],[_,_]).

| ?-  X =  [
     [0,0,0,0,0,0,0,0,0,0,0],
     [100,_,_,_,_,_,_,_,_,_,100],
     [100,_,_,_,_,_,_,_,_,_,100],
     [100,_,_,_,_,_,_,_,_,_,100],
     [100,_,_,_,_,_,_,_,_,_,100],
     [100,_,_,_,_,_,_,_,_,_,100],
     [100,_,_,_,_,_,_,_,_,_,100],
     [100,_,_,_,_,_,_,_,_,_,100],
     [100,_,_,_,_,_,_,_,_,_,100],
     [100,_,_,_,_,_,_,_,_,_,100],
     [100,100,100,100,100,100,100,100,100,100,100]
     ],    laplace(X).
```

```
X=[[0,0,0,0,0,0,0,0,0,0,0],
 [100,51.11,32.52,24.56,21.11,20.12,21.11,24.56,32.52,51.11,100],
 [100,71.91,54.41,44.63,39.74,38.26,39.74,44.63,54.41,71.91,100],
 [100,82.12,68.59,59.80,54.97,53.44,54.97,59.80,68.59,82.12,100],
 [100,87.97,78.03,71.00,66.90,65.56,66.90,71.00,78.03,87.97,100],
 [100,91.71,84.58,79.28,76.07,75.00,76.07,79.28,84.58,91.71,100],
 [100,94.30,89.29,85.47,83.10,82.30,83.10,85.47,89.29,94.30,100],
 [100,96.20,92.82,90.20,88.56,88.00,88.56,90.20,92.82,96.20,100],
 [100,97.67,95.59,93.96,92.93,92.58,92.93,93.96,95.59,97.67,100],
 [100,98.89,97.90,97.12,96.63,96.46,96.63,97.12,97.90,98.89,100],
 [100,100,100,100,100,100,100,100,100,100,100]] ?

yes | ?- laplace([
     [B11, B12, B13, B14],
     [B21, M22, M23, B24],
     [B31, M32, M33, B34],
     [B44, B42, B43, B44]
     ]).

B12 =  -B21 - 4*B31 + 16*M32 - 8*M33 + B34 - 4*B42 + B43,
 B13 =  -B24 + B31 - 8*M32 + 16*M33 - 4*B34 + B42 - 4*B43,
 M22 =  -B31 + 4*M32 - M33 - B42,
 M23 =  -M32 + 4*M33 - B34 - B43 ?

yes
```

4.6 CLP($\mathcal{FD}$) and CLP($\mathcal{N}$)

Gödel's incompleteness theorem gives fundamental limits on integer arithmetic constraints. In order to obtain a decidable constraint language we can either consider the linear fragment, $(\mathbf{N}, 0, 1, +, =)$, which is completely axiomatized by Presburger's arithmetic, or restrict the constraint language on $\mathbf{N}$ given with all its operators.

The later approach is generally undertaken in the implementations of CLP($\mathcal{FD}$) on "finite domains" where the variables are assumed to take their value in finite intervals of the integers. The system CHIP [48] was the first CLP($\mathcal{FD}$) system developed in the mid 80's, following the pioneering work of J.L. Lauriere [34]. CLP($\mathcal{FD}$) systems include in addition to usual arithmetic predicates,

symbolic constraints, e.g.

`element(I,[x1,...,xk],V)` true if $x_I = V$ where I and V are unknowns,

set cardinality constraints, e.g.

`card(N,[X1,...,Xk],V)` true if there are exactly N values equal to V in the list of unknowns $X_1, ..., X_k$,

higher-order cardinality constraints, e.g.

`card(N,[C1,...,Ck])` true if there are exactly N constraints true in the list $C_1, ..., C_k$.

These constraints greatly enhance the expressive power of the constraint language for modeling combinatorial optimization problems[48].

Program 4.20 *One of the simplest example of CLP(FD) program is the N-queens program. The problem, introduced by Gauss in the early days of combinatorics, is to place N queens on an $N \times N$ chess board such that no two queens are placed on a same row, column or diagonal. The CLP(FD) program modelizes the problem with a list of N unknowns which give the line number of each queen in each column (*`domain(L,[1,N])`*). The program places first the inequality constraints between the variables (*`X.=/=Y+d`*), and then enumerates the possible values (*`labeling`*), with* *some heuristics for choosing first the queen with the least domain of possible values ((first-fail heuristics* `ff`*) and trying first the (lines) values in the middle. Solving the 200-queens problem is untractable by pure backtracking, but takes a few seconds with this CLP(FD) program.*

```
queens(N, L) :- list(N, L), domain(L,[1,N]),
                safe(L), labeling(L,ff,middle).

safe([]).
safe([X| Y]) :- noattack(X, Y), safe(Y).

noattack(X, Xs) :- noattack(X, Xs, 1).
noattack(X, [], Nb) :- !.
noattack(X, [Y | Ys], Nb) :- X.=/=Y, X.=/=Y+Nb, X.=/=Y-Nb,
        Nb1 is Nb+1, noattack(X, Ys, Nb1).

list(0, []):- !.
list(N, [ _| L] ) :- M is N-1, list(M, L).

| ?- queens(4,L).

L = [2,4,1,3] ? ;

L = [3,1,4,2] ? ;

no
```

Program 4.21 *[20] The organizers of a congress have 3 rooms and 2 days for eleven half-day sessions (A,B,C,...,K).*

The sessions sets

AJ, JI, IE, CF, FG, DH, BD, KE, BIHG, AGE, BHK, ABCH, DFJ

can't be simultaneous (there exists at least one participant in all the sessions of these sets). Moreover session E has to be given before session J, and the sessions D and F before K.

The organizers have to determine a time-tabling. The problem can be expressed with a simple CLP($\mathcal{FD}$) query.

```
| ?- domain([A,B,C,D,E,F,G,H,I,J,K],[1,4]),
      alldifferent([A,J]),alldifferent([J,I]),alldifferent([I,E]),
      alldifferent([E,C]),alldifferent([C,F]),alldifferent([F,G]),
      alldifferent([D,H]),alldifferent([B,D]),alldifferent([K,E]),
      alldifferent([B,I,H,G]),alldifferent([A,G,E]),
      alldifferent([B,H,K]),alldifferent([A,B,C,H]),
      alldifferent([D,F,J]),
      J.>E, K.>D, K.>F,
      atmost(3,[A,B,C,D,E,F,G,H,I,J,K],1),
      atmost(3,[A,B,C,D,E,F,G,H,I,J,K],2),
      atmost(3,[A,B,C,D,E,F,G,H,I,J,K],3),
      atmost(3,[A,B,C,D,E,F,G,H,I,J,K],4),
      labeling([A,B,C,D,E,F,G,H,I,J,K]).

A = 1, B = 2, C = 4, D = 1, E = 2, F = 2, G = 4, H = 3, I = 1, J = 3, K = 4 ?

yes
```

For practical efficiency reasons, the algorithms for checking the satisfiability of CLP($\mathcal{FD}$) constraints are generally not complete (the satisfiability of the store of constraints is partially checked only), a complete check of satisfiability requires enumeration. The computed answers in CLP($\mathcal{FD}$) systems are thus semi-correct in general 3.3.

The constrained propagation algorithms that are used have for effect to restrict the domain of variables by propagating the constraints at each resolution step, often by a simple reasoning on the bounds of the domain of the variables. The unsatisfiability of the constraints is detected when the domain of a variable becomes empty. The constraints are used to prune the search space concurrently to the logical resolution process. The constraint propagation algorithms used in $CLP(\mathcal{FD})$ originate from Artificial Intelligence, they check the consistency of each constraint separately according to the domain of the variables (arc-consistency).

The constraint propagation algorithms used in CLP($\mathcal{FD}$) can be decribed as particular implementations of a generic algorithm based on few principles. For this purpose let us denote basic constraints by $c, d...$ and constraint systems by $\Gamma, \Gamma', ...$ A variable x will be written with its domain x^d. By abuse of notation, a variable with a singleton domain $x^{\{v\}}$ denotes the value $v \in \mathcal{FD}$ of the domain. As terms, v and $x^{\{v\}}$ are not distinguished.

The set of solutions of a constraint system Γ over $\mathcal{FD}$ is the set of substitutions

$$Sol(\Gamma, \mathcal{FD}) = \{\sigma \mid \sigma = \{x^d \leftarrow v \mid x^d \in V(\Gamma),\ v \in d\},\ \mathcal{FD} \models \Gamma\sigma\}$$

The *reduced domain* of a variable x^d w.r.t. a basic constraint c is the domain

$$DR(x^d, c) = \{v \in d \mid \mathcal{FD} \models \exists(c[v/x^d])\}$$

of values v for which the constraint $c[v/x]$ is satisfiable. A constraint system Γ is *arc-consistent* if

$$\forall c \in \Gamma\ \forall x^d \in V(c)\ DR(x^d, c) = d$$

Definition 4.22 *The* generic constraint propgation algorithm *simplifies a system Γ of constraints over $\mathcal{FD}$ with the following rules associated to basic constraints: "forward checking" (***FC***), "looking-ahead" (***LA***), "partial looking-ahead" (***PLA***) and elimination (***EL***)*

Fail: $c \wedge \Gamma \longrightarrow \bot$
if $x^d \in V(c)$ *and* $DR(x^d, c) = \emptyset$.

FC: $c \wedge \Gamma \longrightarrow \Gamma\sigma$
if $V(c) = \{x^d\}$, $d' = DR(x^d, c)$, $d' \neq \emptyset$, *and* $\sigma = \{x^d \leftarrow y^{d'}\}$ *where* $y \notin V(\Gamma)$.

LA: $c \wedge \Gamma \longrightarrow c\sigma \wedge \Gamma\sigma$
if $|V(c)| > 1$, $x^d \in V(c)$, $d' = DR(x^d, c)$, $d' \neq \emptyset$, $d' \neq d$, $\sigma = \{x^d \leftarrow y^{d'}\}$.

PLA: $c \wedge \Gamma \longrightarrow c\sigma \wedge \Gamma\sigma$

if $|V(c)| > 1$, $x^d \in V(c)$, $DR(x^d, c) \subseteq d' \subset d$, $d' \neq \emptyset$, $\sigma = \{x^d \leftarrow y^{d'}\}$.

EL: $c \wedge \Gamma \longrightarrow \Gamma$ *if* $\mathcal{FD} \models c\sigma$ *for every valuation* σ *of the variables in* c *by values of their domain.*

Lemma 4.23 (Validity) *If* $\Gamma \longrightarrow^*_\sigma \Gamma'$ *then* $Sol(\Gamma, \mathcal{FD}) = \{\sigma\theta \mid \theta \in Sol(\Gamma', \mathcal{FD})\}$.

For instance, disequality contraints `X.=\=Y)` are propagated with the FC rule, symbolic constraints as `element(I,L,V)` are propagated with the LA rule, linear equalities are propagated with the LA rule using a simple reasoning on the bounds of the domain: for a constraint c of the form

$$aX^{[k,l]} \geq bY^{[m,n]} + d,\ a, b > 0,\ d \geq 0$$

we have

$$DR(X^{[k,l]}, c) = [max(k, k'), l]$$
$$DR(Y^{[m,n]}, c) = [m, min(n, n')]$$

where $k' = \lceil \frac{bm+d}{a} \rceil$ and $n' = \lfloor \frac{an-d}{b} \rfloor$. The reduced domain can thus be computed in constant time in this case.

Program 4.24 *Resolution of the puzzle SEND+MORE=MONEY by a CLP($\mathcal{FD}$) program which requires the exploration of at most two choice points (or less according to the choice of the variable to enumerate first).*

```
send(L):-sendc(L), labeling(L).

sendc([S,E,N,D,M,O,R,Y]) :-
     domain([S,E,N,D,M,O,R,Y],[0,9]),
     alldifferent([S,E,N,D,M,O,R,Y]),
     S.=\=0,
     M.=\=0,
                1000*S+100*E+10*N+D
              + 1000*M+100*O+10*R+E
     .= 10000*M+1000*O+100*N+10*E+Y.

| ?- send(L).

L = [9,5,6,7,1,0,8,2] ? ;

no
| ?- sendc([S,E,N,D,M,O,R,Y]).

M = 1, O = 0, S = 9,
Y+90*N.=10*R+D+91*E,
alldifferent([E,N,D,R,Y]),
domain(E,[4,7]),
domain(N,[5,8]),
domain(D,[2,8]),
domain(R,[2,8]),
domain(Y,[2,8]) ?

yes
| ?- sendc([S,E,N,D,M,O,R,Y]),indomain(E).

D = 7, E = 5, M = 1, N = 6,
O = 0, R = 8, S = 9, Y = 2 ? ;

no
| ?- sendc([S,E,N,D,M,O,R,Y]),indomain(R).

M = 1, O = 0, R = 8, S = 9,
```

```
Y+90*N.=D+91*E+80,
alldifferent([E,N,D,Y]),
domain(E,[5,6]),
domain(N,[6,7]),
domain(D,[2,7]),
domain(Y,[2,7]) ? ;

no
```

Constraint propagation is a complete method for some constraints, i.e. it provides a decision procedure. It is the case for instance for systems of inequalities of the form $aX \leq bY + c$ where $a, b, c \geq 0$ [48]. In this case the principle LA is complete and gives a simple decision procedure.

Proposition 4.25 (Completeness of LA) *Let Γ be a constraint system of the form*

$$aX \geq bY + d, \ a, b > 0, \ d \geq 0.$$

Let $\Gamma \longrightarrow_{\sigma}^{*} \Gamma' \not\longrightarrow$. Then Γ is satisfiable if and only if $\Gamma' \neq \bot$, in which case $\{x^{[k,l]} \leftarrow k \mid x \in V(\Gamma')\}$ is a solution.

PROOF: If $\Gamma' = \bot$ then by the validity lemma Γ is insatisfiable. If $\Gamma' \neq \bot$ is irreductible, then for every constraint $c \in \Gamma'$, and every variable $x^d \in V(c)$ we have $d = DR(x^d, c)$. Let $\sigma = \{x^{[k,l]} \leftarrow k \mid x \in V(\Gamma')\}$, we can easily check that each constraint in Γ' is satisfied by σ. Indeed let $aX^{k,l]} \geq bY^{[m,n]} + d$ be a constraint in Γ', by definition of the reduced domain, the constraint $a.k \geq Y^{[m,n]} + d$ is satisfiable, thus $a.k \geq b.m + d$ that is σ is a solution. Therefore $\mathcal{FD} \models \Gamma'\sigma$, and by the validity lemma we get that Γ is satisfiable. □

This class of constraints is important for scheduling problems, as they express precedence constraints, as well as mutual exclusion constraints with a disjonction.

Program 4.26 *Solving by simple CLP($\mathcal{FD}$) queries of a PERT scheduling problem with five tasks A,B,C,D,E, and of a disjunctive scheduling problem where the mutual exclusion constraints between the tasks C and D are treated as Prolog choice point* `;`.

The higher-order predicate `minimize(Goal,Cost)` *computes the optimal solutions to the goal* `Goal` *w.r.t. the objective function* `Cost` *by branch and bound.*

```
  ?- X.>=Y+2.

domain(Y,[0,4294967290]),
domain(X,[2,4294967292]),
X.>=Y+2, t ?

yes | ?- minimize((B.>=A+5,C.>=B+2,D.>=B+3,E.>=C+5,E.>=D+5) , E).

Solution with cost 13

A = 0, B = 5, D = 8,
E = 13,
domain(C,[7,8]),
C.>=5+2 ? ;

no | ?- minimize((B.>=A+5,C.>=B+2,D.>=B+3,E.>=C+5,E.>=D+5,
                  (C.>=D+5 ; D.>=C+5)) , E).
```

```
Solution with cost 18
Solution with cost 17

A = 0, B = 5, C = 7, D = 12, E = 17 ? ;

no
```

Although simple in their principle, constraint propagation algorithms have interesting performances for solving large systems of constraints. Furthermore they can be applied to complex global constraints for which the reduced domains can be computed or approximated by powerful algorithms from Operations Research and graph theory. The CLP($\mathcal{FD}$) programs which have been developed for disjunctive scheduling problems compete today with the best solutions from Operations Research [6]. The reason for this success is the capability of the language to express (and experiment quickly) both complex propagation schemes for global constraints and complex search strategies.

Chapter 5

Formal semantics

The first role of the formal semantics of a programming language is to define mathematically what a program computes. But of course the notion of computation is relative to the choice of the properties of the execution that we wish to observe. We can be interested for instance in the trace of the execution, or in the computed answers (the ordered list of answers or the multi-set or the set), or just in the termination, etc.

A set of *observable properties* (or *observations*) of the execution defines an *equivalence relation on the programs*: $P \equiv P'$ iff for every input, P and P' are observationally undistinguishable. A formal semantics $S(P)$ is correct w.r.t. an equivalence relation on programs $\equiv$, if $S(P) = S(P') \Rightarrow P \equiv P'$, fully abstract if $S(P) = S(P') \Leftrightarrow P \equiv P'$.

The formal semantics can be used for analyzing programs or verifying the soundness of program transformations (for optimizing execution for example). The different ways of defining the formal semantics provide us with different tools for analyzing programs.

In the following section we study the operational semantics of CLP languages, which are based on the definition of the program behavior by an abstract machine (the CSLD resolution rule), then we study for each notion of observable, their related logical, algebraic and fixed point semantics.

5.1 Operational Semantics

For CLP programs, a natural choice of observation from the point of view of theorem proving, is the observation of successes, that is the existence of a CSLD refutation for a goal. We thus define a first equivalence relation $P \equiv_1 P'$ iff for every goal G, G has a CSLD refutation in P iff G has one in P'.

From the point of view of a programming language, we are of course more interested by the set of computed answers to a goal. We can thus define a finer equivalence relation, $P \equiv_2 P'$ iff for every goal G, a constraint c is a computed answer a G in P if and only if c is a computed answer to G in P'.

We could define the operational semantics of a program CLP, as respectively the set of goals which admit a CSLD refutation, and the set of pairs of goals and constraints, $< c, G >$, such that c is a computed answer for G. The lemma of $\wedge$-compositionality 3.7 shows however that the computed answers to a compound goal $(c|A_1, ..., A_n)$, are a simple combination of answers to the atomic goals $(true|A_i)$, $1 \leq i \leq n$. The operational behavior of a CLP program w.r.t. the set of computed answers can thus be entirely caracterized by the set of computed answers to atomic goals only. We can thus define formally the operational semantics of CLP programs for the observation of computed answers by the set of constrained atoms:

$$O_2(P) = \{c|A \mid true|A \longrightarrow^* c|\Box\}$$

Clearly we have $P \equiv_2 P'$ iff $O_2(P) = O_2(P')$.

For the observation of succeses we can define the operational semantics of the program simply as a subset of the $\mathcal{S}$-base:

$$O_1(P) = \{A\rho \in B_\mathcal{S} \mid true|A \longrightarrow^* c|\Box,\ \mathcal{S} \models c\rho\}$$

We have $P \equiv_1 P'$ iff $O_1(P) = O_1(P')$.

5.2 Observation of Successes

In this section we define the fixed point semantics and the logical semantics of CLP programs for the observation of successes, and we show the equivalence with the operational semantics O_1.

Definition 5.1 *Let P be a CLP($\mathcal{S}$) program. The immediate consequence operator $T_P^\mathcal{S} : 2^{B_S} \to 2^{B_S}$ is defined as:*
$T_P^\mathcal{S}(I) = \{A\rho \in B_\mathcal{S}|$ there exists a renamed clause in normal form
$(A \leftarrow c|A_1, ..., A_n) \in P$, and a valuation ρ s.t.
$\mathcal{S} \models c\rho$ and $\{A_1\rho, ..., A_n\rho\} \subseteq I\}\}$

Proposition 5.2 *Let P be a CLP($\mathcal{S}$) program, and I be an $\mathcal{S}$-interpretation. I is a $\mathcal{S}$-model of P if and only if I is a post-fixed point of $T_P^\mathcal{S}$, $T_P^\mathcal{S}(I) \subseteq I$. Furthermore I is a supported $\mathcal{S}$-model of P if and only if I is a fixed point of $T_P^\mathcal{S}$, $T_P^\mathcal{S}(I) = I$.*

Proof: I is a $\mathcal{S}$-model of P,
iff for each clause $A \leftarrow c|A_1, ..., A_n \in P$ and for each $\mathcal{S}$-valuation ρ, if $\mathcal{S} \models c\rho$ and $\{A_1\rho, ..., A_n\rho\} \subseteq I$ then $A\rho \in I$,
iff $T_P^\mathcal{S}(I) \subseteq I$.
I is a fixed point of $T_P^\mathcal{S}$,
iff $T_P^\mathcal{S}(I) = I$,
iff $I = \{A\rho \in I|(A \leftarrow c|A_1, ..., A_n) \in P,\ \mathcal{S} \models c\rho,\ \{A_1\rho, ..., A_n\rho\} \subseteq I\}$
iff I is a supported $\mathcal{S}$-model of P. $\Box$

Proposition 5.3 *$T_P^\mathcal{S}$ is a continuous operator on the lattice of $\mathcal{S}$-interpretations.*

Proof: Let X be a chain of $\mathcal{S}$-interpretations.
$A\rho \in T_P^\mathcal{S}(sup(X))$,
iff $(A \leftarrow c|A_1, ..., A_n) \in P$, $\mathcal{S} \models c\rho$ and $\{A_1\rho, ..., A_n\rho\} \subset sup(X)$,
iff $(A \leftarrow c|A_1, ..., A_n) \in P$, $\mathcal{S} \models c\rho$ and $\{A_1\rho, ..., A_n\rho\} \subset I$, for some $I \in X$,
iff $A \in T_P^\mathcal{S}(I)$ for some $I \in X$,
iff $A \in sup(T_P^\mathcal{S}(X))$. . $\Box$

By the theorem of Knaster-Tarski, the operator $T_P^\mathcal{S}$ has a least fixed point, equal to $T_P^\mathcal{S} \uparrow \omega$, also equal to its least post-fixed point. We can thus define the fixed point semantics of a program CLP($\mathcal{S}$) as the least fixed point of this operator:

$$F_1(P) = lfp(T_P^\mathcal{S}) = T_P^\mathcal{S} \uparrow \omega$$

Theorem 5.4 (Least $\mathcal{S}$-model) *[26] Let P a constraint logic program on $\mathcal{S}$. P has a least $\mathcal{S}$-model, denoted by $M_P^\mathcal{S}$ satisfying:*

$$M_P^\mathcal{S} = F_1(P)$$

PROOF: $F_1(P) = lfp(T_P^{\mathcal{S}})$ is also the least post-fixed point of $T_P^{\mathcal{S}}$, thus by 5.2, $lfp(T_P^{\mathcal{S}})$ is the least $\mathcal{S}$-model of P. □

Theorem 5.5 *[26]* $F_1(P) = O_1(P)$.

PROOF: This result is a corollary of the more general theorem 5.13, given in the following section on the observation of computed constraints. □

5.3 Observation of Computed Constraints

The computed answers of a program CLP($\mathcal{S}$) can also be characterized by a fixed point semantics. The idea is to define an immediate consequence operator on the lattice of constrained atoms. There is a complete adequacy between the computed constraints by CSLD resolution and the constraints associated to atoms in the least fixed point of this operator. This will be used to show a completeness result w.r.t. correct answers of the logical semantics.

Let P a constraint logic program on a structure $\mathcal{S}$ presented by a theory $\mathcal{T}$. A *constrained atom* is a pair $c|A$ composed of a $\mathcal{S}$-satisfiable constraint c and of an atom A containing no function symbol. The set of closed instances of a constrained atom is defined as:

$$[c|A]_{\mathcal{S}} = \{A\rho \mid \mathcal{S} \models c\rho\}$$

The set of constrained atoms forms a complete lattice called the *$\mathcal{T}$-base* and denoted by $B_{\mathcal{T}}$. A constrained interpretation I is a subset of the $\mathcal{T}$-base. We note $[I]_{\mathcal{S}} = \{A\rho \mid c|A \in I,\ S \models c\rho\}$ the $\mathcal{S}$-interpretation associated to I.

Definition 5.6 *The immediate consequence operator $S_P^{\mathcal{S}} : 2^{B_{\mathcal{T}}} \to 2^{B_{\mathcal{T}}}$ is defined as:*
$S_P^{\mathcal{S}}(I) = \{c|A \in B_{\mathcal{T}} \mid$ *there exists a renamed clause in normal form*
$(A \leftarrow d|A_1, ..., A_n) \in P$, *and constrained atoms* $\{c_1|A_1, ..., c_n|A_n\} \subseteq I$,
s.t. $c = d \wedge \bigwedge_{i=1}^{n} c_i$ *is $\mathcal{S}$-satisfiable*$\}$.

Exercise 5.7 *Show that $S_P^{\mathcal{S}}$ is a continuous operator on the lattice of constrained interpretations.*

Definition 5.8 *The fixed point semantics of a program CLP($\mathcal{S}$) is defined as the least fixed point of $S_P^{\mathcal{S}}$,*

$$F_2(P) = lfp(S_P^{\mathcal{S}}) = S_P^{\mathcal{S}} \uparrow \omega.$$

Example 5.9 *Consider the CLP($\mathcal{H}$)* `append` *program*

```
append(A,B,C):- A=[], B=C.
append(A,B,C):- A=[X|L], C=[X|R], append(L,B,R).
```

The iteration of the non-ground immediate consequence operator from the empty constrained interpretation enumerates the CSLD answer constraints to the goal `append(A,B,C)`*:*

$$\begin{array}{ll}
S_P^{\mathcal{H}} \uparrow 0 & = \emptyset \\
S_P^{\mathcal{H}} \uparrow 1 & = \{A = [], B = C|append(A, B, C)\} \\
S_P^{\mathcal{H}} \uparrow 2 & = S_P^{\mathcal{H}} \uparrow 1 \cup \\
 & \quad \{A = [X|L], C = [X|R], L = [], B = R|append(A, B, C)\} \\
 & = S_P^{\mathcal{H}} \uparrow 1 \cup \{A = [X], C = [X|B]|append(A, B, C)\} \\
S_P^{\mathcal{H}} \uparrow 3 & = S_P^{\mathcal{H}} \uparrow 2 \cup \{A = [X, Y], C = [X, Y|B]|append(A, B, C)\} \\
S_P^{\mathcal{H}} \uparrow 4 & = S_P^{\mathcal{H}} \uparrow 3 \cup \{A = [X, Y, Z], C = [X, Y, Z|B]|append(A, B, C)\} \\
... & = ...
\end{array}$$

Lemma 5.10 *For every constrained interpretation I, $[S_P^{\mathcal{S}}(I)]_{\mathcal{S}} = T_P^{\mathcal{S}}([I]_{\mathcal{S}})$.*

PROOF: We prove the two inclusions separately.

Let $c|A \in S_P^{\mathcal{S}}(I)$ and ρ be a valuation solution of c. By definition of $S_P^{\mathcal{S}}$ there exists a renamed clause in normal form $(A \leftarrow d|A_1, ..., A_n) \in P$ and constrained atoms $\{c_1|A_1, ..., c_n|A_n\} \subseteq I$, such that $c = d \wedge \bigwedge_{i=1}^n c_i$. Thus ρ is also a solution of $c_1, ..., c_n$. Therefore $\{A_1\rho, ..., A_n\rho\} \subseteq [I]_{\mathcal{S}}$ and by definition of $T_P^{\mathcal{S}}$, we have $A\rho \in T_P^{\mathcal{S}}$.

In the other direction, let $A\rho \in T_P^{\mathcal{S}}([I]_{\mathcal{S}})$. By definition of $T_P^{\mathcal{S}}$, there exists a clause $(A \leftarrow d|A_1, ..., A_n) \in P$ such that $A_1\rho, ..., A_n\rho\} \subseteq [I]_{\mathcal{S}}$ and ρ is solution of d. By definition of $[I]_{\mathcal{S}}$, there exist constrained atoms $\{c_1|A_1, ..., c_n|A_n\} \subseteq I$ renamed in such a way as ρ is a solution of $c_1, ..., c_n$. Let $c = d \wedge \bigwedge_{i=1}^n c_i$, ρ is a solution of c, thus c is $\mathcal{S}$-satisfiable and by definition of $S_P^{\mathcal{S}}$, we have $c|A \in S_P^{\mathcal{S}}(I)$. Therefore $A\rho \in [S_P^{\mathcal{S}}(I)]_{\mathcal{S}}$. □

Theorem 5.11 *[26] For every ordinal α, $T_P^{\mathcal{S}} \uparrow \alpha = [S_P^{\mathcal{S}} \uparrow \alpha]_{\mathcal{S}}$.*

PROOF: The proof is by transfinite induction on α.

The base case $\alpha = 0$ is trivial.

For a successor ordinal, we have
$[S_P^{\mathcal{S}} \uparrow \alpha] = [S_P^{\mathcal{S}}(S_P^{\mathcal{S}} \uparrow \alpha - 1)]_{\mathcal{S}}$,
$= T_P^{\mathcal{S}}([S_P^{\mathcal{S}} \uparrow \alpha - 1]_{\mathcal{S}})$ by lemma 5.10,
$= T_P^{\mathcal{S}}(T_P^{\mathcal{S}} \uparrow \alpha - 1)$ by induction,
$= T_P^{\mathcal{S}} \uparrow \alpha$.

For a limit ordinal, we have
$[S_P^{\mathcal{S}} \uparrow \alpha]_{\mathcal{S}} = [\bigcup_{\beta<\alpha} S_P^{\mathcal{S}} \uparrow \beta]_{\mathcal{S}}$
$= \bigcup_{\beta<\alpha} [S_P^{\mathcal{S}} \uparrow \beta]_{\mathcal{S}}$,
$= \bigcup_{\beta<\alpha} T_P^{\mathcal{S}} \uparrow \beta$ by induction,
$= T_P^{\mathcal{S}} \uparrow \alpha$. □

Corollary 5.12 *For every integer $n \geq 0$, $T_P^{\mathcal{S}} \uparrow n$ has a finite presentation.*

PROOF: For every integer n, $S_P^{\mathcal{S}} \uparrow n$ is finite and $[S_P^{\mathcal{S}} \uparrow n]_{\mathcal{S}} = T_P^{\mathcal{S}} \uparrow n$. □

Theorem 5.13 (Full abstraction) *[18] $O_2(P) = F_2(P)$.*

PROOF: If c is a computed answer for the goal $true|A$, we show that $c|A \in S_P^{\mathcal{S}} \uparrow \omega$ by induction on the length of the derivation m.

The base case $m = 1$ corresponds to the resolution of the goal by a fact of the form $A \leftarrow c$. We have $c|A \in S_P^{\mathcal{S}} \uparrow 1$.

For the induction step, the derivation is of the form:

$$(true|A) \longrightarrow (d|A_1, ..., A_n) \longrightarrow^* (c|\Box).$$

By the $\wedge$-compositionality lemma there exist computed answers $c_1, ..., c_n$ for the goals $A_1, ..., A_n$, such that $c = d \wedge \bigwedge_{i=1}^n c_i$. By the induction hypothesis there exist $c_1|A_1, ..., c_n|A_n \in S_P^{\mathcal{S}} \uparrow \omega$. Thus by definition of $S_P^{\mathcal{S}}$ we obtain $c|A \in S_P^{\mathcal{S}} \uparrow \omega + 1 = S_P^{\mathcal{S}} \uparrow \omega$.

In the other direction, if $c|A \in S_P^{\mathcal{S}} \uparrow n$, we show by induction on n that c is a computed answer for the goal $true|A$. The base case $n = 1$ is equivalent to the previous base case.

For the induction step, by definition of $S_P^{\mathcal{S}}$, there exists a renamed clause in normal form $(A \leftarrow d|A_1, ..., A_n) \in P$ and $\{c_1|A_1, ..., c_n|A_n\} \subseteq S_P^{\mathcal{S}} \uparrow n - 1$ such that $c = d \wedge \bigwedge_{i=1}^n$ is $\mathcal{S}$-satisfiable.

We thus have the first step of resolution:

$$(true|A) \longrightarrow (d|A_1, ..., A_n)$$

and by induction, for every i, $1 \leq i \leq n$:

$$(true|A_i) \longrightarrow^* (c_i|\Box).$$

As c is satisfiable then by lemma 3.7, we deduce that

$$(true|A) \longrightarrow^* (c|\Box).$$

$\Box$

Corollary 5.14 *c is a computed answer for the goal $d|A_1, ..., A_n$ if and only if there exists $\{c_1|A_1, ..., c_n|A_n\} \subseteq S_P^S \uparrow \omega$ such that $c = d \wedge \bigwedge_{i=1}^n c_i$.*

PROOF: By the $\wedge$-compositionality lemma 3.7. $\Box$

$S_P^S \uparrow \omega$ captures the set of computed answer constraints with program P, nevertheless this set may be infinite and it may contain too much information for proving some properties of the program. *Abstract interpretation* [5] is a method for proving properties of programs without handling irrelevant information. The idea is to replace the real computation domain by an abstract computation domain which retains sufficient information w.r.t. the property to prove.

Example 5.15 (Groundness analysis by abstract interpretation) *Let us consider the CLP($\mathcal{H}$) append program in 5.9, and let us infer information about the groundness of the arguments of append after a success. More precisely let us ask the following question: what is the groundness relation between arguments after a success in append?*

The term structure can be abstracted by a boolean structure which expresses the groundness of the arguments. We thus associate a CLP(Bool) abstract program *by abstracting equality constraints over Herbrand variables by boolean constraints representing the groundness of the variables:*

```
append(A,B,C):- A=true, B=C.
append(A,B,C):- A=X/\L, C=X/\R, append(L,B,R).
```

The least fixed point of the immediate consequence operator, computed in at most 2^3 steps, expresses the groundness relation between arguments of the concrete program.

$$\begin{array}{ll}
S_P^{Bool} \uparrow 0 & = \emptyset \\
S_P^{Bool} \uparrow 1 & = \{A = true, B = C|append(A,B,C)\} \\
S_P^{Bool} \uparrow 2 & = S_P^{Bool} \uparrow 1 \cup \\
& \{A = X \wedge L, C = X \wedge R, L = true, B = R|append(A,B,C)\} \\
& = S_P^{Bool} \uparrow 1 \cup \{C = A \wedge B|append(A,B,C)\} \\
S_P^{Bool} \uparrow 3 & = S_P^{Bool} \uparrow 2 \cup \\
& \{A = X \wedge L, C = X \wedge R, R = X \wedge B|append(A,B,C)\} \\
& = S_P^{Bool} \uparrow 2 \cup \{C = A \wedge B|append(A,B,C)\} \\
& = S_P^{Bool} \uparrow 2 = S_P^{Bool} \uparrow \omega
\end{array}$$

In a success of $append(A,B,C)$ C is ground if and only if A and B are ground.

Example 5.16 (Groundness analysis of reverse) *Concrete CLP($\mathcal{H}$) program:*

```
rev(A,B) :- A=[], B=[].
rev(A,B) :- A=[X|L], rev(L,K), append(K,[X],B).
```

Abstract CLP(Bool) program:

```
rev(A,B) :- A=true, B=true.
rev(A,B) :- A=X/\L, rev(L,K), append(K,X,B).
```

$$\begin{aligned}
S_P^{Bool} \uparrow 0 &= \emptyset \\
S_P^{Bool} \uparrow 1 &= \{A = true, B = true | rev(A,B)\} \\
S_P^{Bool} \uparrow 2 &= S_P^{Bool} \uparrow 1 \cup \{A = X, B = X | rev(A,B)\} \\
&= S_P^{Bool} \uparrow 1 \cup \{A = B | rev(A,B)\} \\
S_P^{Bool} \uparrow 3 &= S_P^{Bool} \uparrow 2 \cup \{A = X \wedge L, L = K, B = K \wedge X | rev(A,B)\} \\
&= S_P^{Bool} \uparrow 2 \cup \{A = B | rev(A,B)\} = S_P^{Bool} \uparrow 2 = S_P^{Bool} \uparrow \omega
\end{aligned}$$

The fixpoint semantics is also useful to link the operational semantics of CLP programs to their logical semantics.

Theorem 5.17 (Soundness of CSLD resolution) *[26] Let P be a $CLP(\mathcal{S})$ program. If c is a computed answer for the goal G then c is a correct answer.*

PROOF: If $G = (d|A_1, ..., A_n)$, we deduce from the $\wedge$-compositionality lemma 3.7,that there exist computed answers $c_1, ..., c_n$ for the goals $A_1, ..., A_n$ such that $c = d \wedge \bigwedge_{i=1}^{n} c_i$ is satisfiable. For every i, $1 \leq i \leq n$ we have
$c_i|A_i \in S_P^{\mathcal{S}} \uparrow \omega$, by 5.13,
$[c_i|A_i]_{\mathcal{S}} \subseteq M_P^{\mathcal{S}}$, by 5.11, and 5.2,
$P \models_{\mathcal{S}} \forall(c_i \supset A_i)$ as $M_P^{\mathcal{S}}$ is the least $\mathcal{S}$-model of P,
$P \models_{\mathcal{S}} \forall(c \supset A_i)$ as $\mathcal{S} \models \forall(c \supset c_i)$.
Therefore we have $P \models_{\mathcal{S}} \forall(c \supset (d \wedge A_1 \wedge ... \wedge A_n))$. □

Theorem 5.18 (Completeness of CSLD resolution) *[36] Let P be a $CLP(\mathcal{S})$ program. If c is a correct answer for the goal G then there exists a (possibly infinite) set $\{c_i\}_{i \geq 0}$ of computed answers for G, such that:*

$$\mathcal{S} \models \forall(c \supset \bigvee_{i \geq 0} \exists Y_i c_i).$$

PROOF: According to the $\wedge$-compositionality lemma 3.7, it is sufficient to prove the theorem for an atomic goal A. Let c be a correct answer for the goal A. For every solution ρ of c,
$A\rho$ is true in all the $\mathcal{S}$-models of P,
iff $A\rho$ is true in the least $\mathcal{S}$-model of P,
iff $A\rho \in T_P^{\mathcal{S}} \uparrow \omega$, by 5.5,
iff $A\rho \in [S_P^{\mathcal{S}} \uparrow \omega]_{\mathcal{S}}$, by 5.10,
iff $c_\rho | A \in S_P^{\mathcal{S}} \uparrow \omega$, for some constraint c_ρ s.t. ρ is solution of $\exists Y_\rho c_\rho$, where $Y_\rho = V(c_\rho) \setminus V(A)$,
iff c_ρ is a computed answer for A (by 5.13).
By taking the collection of all these constraints c_ρ we obtain:

$$\mathcal{S} \models \forall(c \supset \bigvee_{c_\rho} \exists Y_\rho c_\rho)$$

□

The fact that a possibly infinite set of computed answers has to be considered to insure the completeness w.r.t. correct answers in the structure $\mathcal{S}$, is the same as for logic programs without constraints, when the Herbrand's domain, $\mathcal{H}$, is formed on a finite alphabet. For instance if $S_F = \{0, s\}$, then with the program
$P = \{p(0),\ p(s(X)) \leftarrow p(X)\}$

the goal $p(X)$ has an infinite set of successful derivations with set of computed substitutions

$$\{X \leftarrow s^i(0) \mid i \geq 0\}.$$

We have $P, \mathcal{H} \models \forall X p(X)$, but the identity substitution is not a computed answer.

If we take for the notion of correct answers, not the truth in the structure $\mathcal{S}$, but the truth w.r.t. the logical consequences of the theory $\mathcal{T}$ of presentation of $\mathcal{S}$, the number of computed answers to consider is finite.

Theorem 5.19 (Completeness w.r.t. the theory of the structure) *[36] Let P be a constraint logic program on a structure $\mathcal{S}$ presented by a theory $\mathcal{T}$. If*

$$P, \mathcal{T} \models \forall(c \supset G) \wedge \exists(c)$$

then there exists a finite set $\{c_1, ..., c_n\}$ of computed answers to G, such that:

$$\mathcal{T} \models \forall(c \supset \exists Y_1 c_1 \vee ... \vee \exists Y_n c_n).$$

Proof: If $P, \mathcal{T} \models c \supset G$ then for every model $\mathcal{S}$ of $\mathcal{T}$, for every $\mathcal{S}$-solution ρ of c, there exists a computed constraint $c_{\mathcal{S},\rho}$ for G s.t. $\mathcal{S} \models c_{\mathcal{S},\rho}\rho$. Let $\{c_i\}_{i \geq 0}$ be the set of these computed answers.

Then for every model $\mathcal{S}$ and for every $\mathcal{S}$-valuation ρ, $\mathcal{S} \models c \supset \vee_{i \geq 1} \exists Y_i c_i$, therefore $\mathcal{T} \models c \supset \vee_{i \geq 1} \exists Y_i c_i$, hence by applying the compactness theorem of first-order logic, there exists a finite part, let $\{c_i\}_{1 \leq i \leq n}$, such that $\mathcal{T} \models c \supset \vee_{i=1}^{n} \exists Y_i c_i$. □

5.4 Observation of Finite Failures

Definition 5.20 *Let P a program CLP($\mathcal{S}$). A* derivation CSLD is fair *if every atom which appears in a goal of the derivation is selected after a finite number of resolution steps.*

A fair CSLD tree *for a goal G is a CSLD derivation tree for G in which all derivations are fair.*

A goal G i finitely failed *if G has a fair CSLD derivation tree to G, which is finite and which contains no success.*

Finite failure is another observable property of logic programs that it is worth considering in addition to computed answers. Finite failure corresponds to a notion of negative answer to a goal. However the logical semantics based on the logical consequences of the program where each rule is viewed as an implication doesn't allow us to infere negative logical consequences, just because the Herbrand's base constitute a model of the program in which all the atoms are true.

On the other hand, the declarative semantics based on the least $\mathcal{S}$-model of the program is undecidable. This is easy to see on the Herbrand's domain with a Prolog program. Indeed, let us suppose the opposite, as Prolog is a language universal, there thus exists a Prolog program for defining the following predicates:

`success(P,B)` which is true if $M_P \models \exists B$ (i.e. if the goal B has a successful SLD derivation with the program P), false otherwise (i.e. $M_P \models \neg\exists B$),

`fail(P,B)` the negation of `success(P,B)`.

We obtain a contradiction by considering the following program and goal:

```
loop:- loop.

contr(P):- success(P,P), loop.
contr(P):- fail(P,P).

?- contr(contr).
```

If `contr(contr)` has an SLD refutation, then it is also the case for the goal `success(contr,contr)` which is true, hence `fail(contr,contr)` fails, thus by definition of the predicate `contr`, the goal `contr(contr)` doesn't admit an SLD refutation: a contradiction.

If `contr(contr)` admits a successful derivation, then the goal `fail(contr,contr)` is true, thus the goal has an SLD refutation: a contradiction.

Hence we conclude that the programs `success` and `fail` can not exist.

In order to give a declarative semantics to finite failures, it is thus necessary to review the logical interpretation of the program, and to read the rules of the program, as definitions of the predicates by equivalences, and instead of by implications.

Definition 5.21 *Let P be a CLP program on a structure $\mathcal{S}$, presented by a theory $\mathcal{T}$. The* Clark's completion *of P is the set of formulas formed of $\mathcal{T}$ and of P^* defined as the set of formulas of the form*

$$\forall X p(X) \leftrightarrow (\exists Y_1 c_1 \wedge A_1^1 \wedge ... \wedge A_{n_1}^1) \vee ... \vee (\exists Y_k c_k \wedge A_1^k \wedge ... \wedge A_{n_k}^k)$$

obtained for each predicate symbol $p \in P$ by collecting the rules which define p in P, $p(X) \leftarrow c_i | A_1^i, ..., A_{n_i}^i$ with local variables Y_i,

or of the form

$$\forall X \neg p(X)$$

if p is not defined in P.

Example 5.22 *Let P be the program $CLP(\mathcal{H})$ defined by the only rule*

$$p(s(X) \leftarrow p(X)$$

i.e. $p(X) \leftarrow X = s(Y)|p(Y)$. The Clark's completion of P is the equality theory CET augmented with

$$P^* = \{\forall x\ p(x) \leftrightarrow \exists y\ x = s(y) \wedge p(y).$$

The goal $p(0)$ is finitely failed, we verify easily that $P^*, CET \models \neg p(0)$. On the other hand the goal $p(X)$ has an infinite fair derivation, it is thus not finitely failed, hence $P^*, CET \not\models \neg \exists x p(x)$. The cause of this situation is the existence of non-standard models of CET (cf. 4.8), on the other hand in the standard model $P^*, \mathcal{H} \models \neg \exists x p(x)$.

We shall show that the logical consequences of the program's completion does characterize finite failures. Before that we show that the Clark's completion doesn't change the logical semantics of correct answers.

Proposition 5.23 *Let P be a constraint logic program on a structure $\mathcal{S}$ and I be an $\mathcal{S}$-interpretation on P. The following propositions are equivalent:*

i) I is a supported $\mathcal{S}$-model of P, *ii) I is a $\mathcal{S}$-model of P^*.*

iii) I is a fixed point of $T_P^{\mathcal{S}}$

PROOF: I is a $\mathcal{S}$-model of P
iff I is a $\mathcal{S}$-model of $\forall X\ p(X) \leftarrow \phi_1 \vee ... \vee \phi_k$ for every formula $\forall X\ p(X) \leftrightarrow \phi_1 \vee ... \vee \phi_k$ in P^*,
iff I is a post-fixed point of $T_P^{\mathcal{S}}$, i.e. $.T_P^{\mathcal{S}}(I) \subseteq I$.
I is a supported $\mathcal{S}$-interpretation of P,
iff I is a $\mathcal{S}$-model of $\forall X\ p(X) \rightarrow \phi_1 \vee ... \vee \phi_k$ for every formula $\forall X\ p(X) \leftrightarrow \phi_1 \vee ... \vee \phi_k$ in P^*,
iff I is a pre-fixed point of $T_P^{\mathcal{S}}$, i.e. $I \subseteq T_P^{\mathcal{S}}(I)$.
We deduce that I is a supported $\mathcal{S}$-model of P,
iff I is a $\mathcal{S}$-model of P^*,
iff I is a fixed point of $T_P^{\mathcal{S}}$. □

Theorem 5.24 *Let P be a constraint logic program on a structure $\mathcal{S}$.*

i) P^* has the same least $\mathcal{S}$-model than P, $M_P^{\mathcal{S}} = M_{P^*}^{\mathcal{S}}$,

ii) $P \models_{\mathcal{S}} c \rightarrow A$ iff $P^* \models_{\mathcal{S}} c \rightarrow A$, for every constraint c and every atom A,

iii) $P, \mathcal{T} \models c \rightarrow A$ iff $P^*, \mathcal{T} \models c \rightarrow A$.

PROOF: i) follows immediately from 5.5 and 5.23.
For iii) we clearly have $(P, \mathcal{T} \models c \rightarrow A) \Rightarrow (P^*, \mathcal{T} \models c \rightarrow A)$. We show the contrapositive of the opposite, $(P, \mathcal{T} \not\models c \rightarrow A) \Rightarrow (P^*, \mathcal{T} \not\models c \rightarrow A)$.
Let I be a model of P and T, based on a structure $\mathcal{S}$, let ρ be a valuation such that $I \models \neg A\rho$ and $\mathcal{S} \models c\rho$.
We have $M_P^{\mathcal{S}} \models \neg A\rho$, thus $M_{P^*}^{\mathcal{S}} \models \neg A\rho$, and as $\mathcal{T} \models c\rho$, we conclude that $P^*, \mathcal{T} \not\models c \rightarrow A$.
The proof of ii) is identical, the structure $\mathcal{S}$ being fixed. □

Remark 5.25 *As shown by the completeness theorems 5.18, and 5.19, $P \models_{\mathcal{S}} c \rightarrow A$ doesn't imply $P, \mathcal{T} \models c \rightarrow A$. The previous theorem shows that the replacement of P by P^* exactly preserves these differences for the logical consequences of the form $c \rightarrow A$.*

Theorem 5.26 (Soundness of the negation by finite failure) *Let P be a logic program with constraint on a structure $\mathcal{S}$ presented by a theory $\mathcal{T}$. If G is finitely failed then $P^*, \mathcal{T} \models \neg G$.*

PROOF: By induction on the height h of the tree in finite failure for $G = c|A, \alpha$ where A is the selected atom at the root of the tree.
In the base case $h = 1$, the constrained atom $c|A$ has no CSLD transition, we can deduce that $P^*, \mathcal{T} \models \neg(c \wedge A)$ hence that $P^*, \mathcal{T} \models \neg G$.
For the induction step, let us suppose $h > 1$. Let $G_1, ..., G_n$ be the sons of the root and $Y_1, ..., Y_n$ be the respective sets of introduced variables. We have $P^*, \mathcal{T} \models G \leftrightarrow \exists Y_1\ G_1 \vee ... \vee \exists_n\ G_n$. By induction hypothesis, $P^*, \mathcal{T} \models \neg G_i$ for every $1 \leq i \leq n$, therefore $P^*, \mathcal{T} \models \neg G$. □

Lemma 5.27 *If $(c|A) \longrightarrow (d|A_1, ..., A_n)$ then $[d|A]_{\mathcal{S}} \subseteq T_P^{\mathcal{S}}([\{d|A_1, ..., d|A_n\}]_{\mathcal{S}})$.*

Theorem 5.28 (Completeness of finite failure) *[26] Let P be a constraint logic program on a structure $\mathcal{S}$ presented by a theory $\mathcal{T}$. If $P^*, \mathcal{T} \models \neg G$ then G is finitely failed.*

PROOF: We show that if G has a fair CSLD tree which is not finitely failed then $P^*, \mathcal{T}, \exists(G)$ is satisfiable.

If G has a succesful derivation then by the soundness theorem (5.17), $P^*, \mathcal{T} \models \exists G$. Otherwise G has a fair infinite CSLD-derivation

$$G = c_0|G_0 \longrightarrow c_1|G1 \longrightarrow c_2|G2 \longrightarrow ...$$

For every $i \geq 0$, c_i is $\mathcal{T}$-satisfiable, thus by the compactness theorem of first-order logic, $c_\omega = \bigcup_{i\geq 0} c_i$ is $\mathcal{T}$-satisfiable.

Let $\mathcal{S}$ be a model of $\mathcal{T}$ s.t. $\mathcal{S} \models \exists(c_\omega)$. Let $I_0 = \{A\rho \mid A \in G_i$ for some $i \geq 0$ and let $\mathcal{S} \models c_\omega\rho\}$. As the derivation is fair, every atom A in I_0 is selected in a step of resolution, thus $c_\omega|A \longrightarrow c_\omega|A_1, ..., A_n$ with $[c_\omega|A]_\mathcal{S} \cup ... \cup [c_\omega|A_n]_\mathcal{S} \subseteq I_0$. Of the lemma 5.27 we deduce that $I_0 \subseteq T_P^\mathcal{S}(I_0)$ hence $I_0 \subseteq T_P^\mathcal{S}(I_0)$.

By the theorem of Knaster-Tarski, the iterated application up to ordinal ω of the operator $T_P^\mathcal{S}$ from I_0 leads to a fixed point I s.t. $I_0 \subseteq I$, thus $[c_\omega|G_0]_\mathcal{S} \in I$. We deduce that $P^*, \exists(G)$ is $\mathcal{S}$-satisfiable, thus that $P^*, \mathcal{T}, \exists(G)$ is satisfiable. $\square$

Introducing the connective of negation in logic programs, suppresses the restriction to Horn clause formulas, and generalizes the approach to the whole first-order logic. The Clark's completion of logic programs with negation can be inconsistent however, e.g. $p \leftarrow \neg p$.. One solution to restore the consistency of such programs is to skip to Kleene's three-valued logic [17], [33]. The principle of negation by finite failure is correct but incomplete w.r.t. the three-valued logic semantics of logic programs with negation. Another principle called of constructive negation is proved complete for CLP programs with negation in [46]. In [13] we define a principle of constructive negation by pruning, where negation is handled by a concurrent mechanism of pruning between standard CSLD derivation trees, and whose computed answers are characterized by a simple fixed point semantics. These results make it possible to investigate the implementation of CLP systems not limited to Horn clausal formulas, in which for instance the implementation of the optimization predicates (cf. 4.26) can be derived [13].

Bibliography

[1] K.R. Apt, *Logic Programming*, Handbook of Theoretical Computer Science, J. van Leeuwen ed., Elsevier, pp.493-574 (1990).

[2] K.R. Apt, H.A. Blair, A. Walker, *Towards a theory of declarative knowledge*, in Foundations of deductive databases and logic programming, Minker, J. (ed.), Morgan Kaufmann, Los Altos (1987).

[3] K.R. Apt, M.H. van Emden, *Contributions to the theory of logic programming*, JACM, 29(3), pp.841-862 (1982).

[4] M. Carlsson et al., *Sicstus-Prolog reference manual V3*, Technical report, Swedish Institute in Computer Science, (1996).

[5] P. Cousot, R. Cousot, *Abstract interpretation and application to logic programs*, Journal of Logic Programming, 13(2 et 3), pp.103-179 (1992).

[6] Y. Caseau, F. Laburthe, *Improved CLP scheduling with tasks intervals*, Proc. International Conference on Logic Programming, ICLP'94, Santa Margharita Ligure, MIT Press (1994).

[7] V. Chvatal, *Linear programming*, W. H. Freeman and Co, 478pp. (1983)

[8] K.L. Clark, *Negation as Failure*, in Logic and Databases, Ed. H. Gallaire and J. Minker, Plenum Pub. (1978).

[9] A. Colmerauer, *Prolog II: Reference manual and theoretical model*, Rapport GIA, Univ. Marseille. (1982).

[10] A. Colmerauer : "Opening the Prolog-III universe", Byte, August 1987.

[11] B. Courcelle, *Fundamentals properties of infinite trees*, Theoretical Computer Science, 25(2), pp.95-169, (1983).

[12] M. Dincbas, H. Simonis and P. Van Hentenryck : "Solving large combinatorial problems in Logic Programming", ECRC technical report TR-LP-21, 1987, and Journal of Logic Programming, 8(1-2), pp.74-94, (1990).

[13] F. Fages, *Constructive negation by pruning*, Journal of Logic Programming, 32(2), pp.85-118, August 1997.

[14] F. Fages, *Programmation logique par contraintes*, Ellipses, Paris, 192p., 1996.

[15] F. Fages, P. Ruet, S. Soliman, *Linear concurrent constraint programming: operational and phase semantics*, Information and Computation, 165(1), February 2001.

[16] M. Falaschi, G. Levi, M. Martelli, C. Palamidessi, *A model-theoretic reconstruction of the operational semantics of logic programs*, Information and Control 103, pp.86-113 (1993).

[17] M. Fitting, *A Kripke/Kleene semantics for logic programs*, Journal of Logic Programming, 2(4), pp.295-312 (1985)

[18] M. Gabbrielli, G. Levi, *Modeling answer constraints in constraint logic programs*, Proc. International Symposium on Logic Programming ICLP'91, Paris, MIT Press pp.238-252, (1991).

[19] W.D. Goldfarb, *The undecidability of the second-order unification problem*, Theoretical Computer Science, Vol. 13, pp. 225-230 (1981).

[20] M. Gondran, M. Minoux, *Graphes et algorithmes*, Collection de la Direction des Études et Recherches D'Électricité de France, 37, (1995).

[21] Gotha, *Les problèmes d'ordonnancement*, Recherche Opérationnelle/Operations Research, 27(1), pp.77-150 (1993).

[22] J. Herbrand, *Recherches sur la théorie de la démonstration*, Thèse de doctorat (1930), in Ecrits logiques, PUF (1968).

[23] H. Hong, *RISC-CLP(Real): logic programming with non-linear constraints over the Reals*, in "Constraint logic programming : selected research", Ed. F. Benhamou and A. Colmerauer ed., MIT Press, 1993.

[24] G.P. Huet, *Constrained resolution: a complete method for higher order logic*, Ph. D. thesis, Case Western Reserve Univ (1972).

[25] G. Huet, *Résolution d'équation dans les langages d'ordre 1, 2, ... omega*, Thèse d'état, Univ. d'Orsay (1976).

[26] J. Jaffar and J-L. Lassez, *Constraint Logic Programming*, Research Report, University of Melbourne, 1986. Also in the proceedings of POPL'87 (1987).

[27] J. Jaffar and S. Michaylov, *Methodology and implementation of a CLP system*, in Proc. 4th Internation Conference on Logic Programming, pp.196-218, Cambridge, MIT Press, (1987).

[28] J. Jaffar and M. Maher, *Constraint Logic Programming: a survey*, Journal of Logic Programming, 19-20, (1994).

[29] J.P. Jouannaud, C. Kirchner, *Solving equations in abstract algebras: a rule-based survey of unification*, in Computational logic, J.L. Lassez and G. Plotkin ed., MIT Press (1991).

[30] D. Kapur, P. Narendran, *Complexity of unification problems with associative-commutative operators*, Journal of Automated Reasoning, 9, pp. 261-288 (1992).

[31] R. Kowalski, *Predicate Logic as Programming Language*, Information Processing 74, pp.569-574, (1974).

[32] R. Kowalski, *Logic for Problem Solving*, North Holland (1979).

[33] K. Kunen, *Negation in logic programming*, Journal of Logic Programming, 4(3), pp.289-308, (1987).

[34] J.L. Lauriere, *A language and a program for stating and solving combinatorial problems*, Artificial Intelligence 10, pp.29-127, (1978).

[35] J.W. Lloyd, *Foundations of Logic Programming*, Springer Verlag (1987).

[36] M. Maher, *Logic semantics for a class of comitted choice languages*, Proc. 4th ICLP, MIT Press, pp.858-876 (1987).

[37] M. Maher, *Equivalences of logic programs*, in Foundations of Deductive Databases and Logic Programming, Morgan Kaufman, pp.627-658, (1988).

[38] M. Maher, *Complete axiomatizations of the algebras of finite, rational and infinite trees*, Proc 3rd Symp. on Logic in Computer Science, Edinburgh, pp.348-357, (1988).

[39] D. Miller, *A logic programming language with λ-abstraction, function variables and simple unification*, Journal of Logic and Computation, 1(4), pp.497-536 (1991).

[40] R. Mohr and T.C. Henderson, "Arc and path consistency revisited", Artificial Intelligence, 28:225-233, 1986.

[41] W. Older, A. Vellino, *Constraint arithmetic on real intervals*, in "Constraint logic programming : selected research", Ed. F. Benhamou and A. Colmerauer ed., MIT Press, 1993.

[42] J.A. Robinson, *A machine-oriented logic based on the resolution principle*, JACM 12, 1, pp.23-41 (1965).

[43] V.J. Saraswat : "Concurrent Constraint Programming Languages", MIT Press series in Logic Programming, (1993).

[44] J.R. Shoenfield, *Mathematical logic*, Addison-Wesley Pub., 1967.

[45] L. Sterling, E. Shapiro, *The Art of Prolog*, MIT Press (1986).

[46] P. Stuckey, *Constructive negation for constraint logic programming*, Proc. LICS'91 ACM, (1991).

[47] M.H. van Emden, R.H. Kowalski, *The semantics of predicate logique as a programming language*, JACM, 23(4), pp.733-742 (1976).

[48] P. Van Hentenryck : "Constraint Satisfaction in Logic Programming", MIT Press 1989.

[49] P. Van Hentenryck and Y. Deville : "Efficient Arc Consistency Algorithm for a class of CSP Problems", proc. IJCAI 91, Sidney, 1991.

[50] P. Van Hentenryck, V. Saraswat, Y. Deville, *Design, implementation and evaluation of the constraint language CC($\mathcal{FD}$)*, in Constraint Programming: basics and trends, A. Podelski Ed., Châtillon-sur-Seine, Springer-Verlag LNCS 910, pp.68-90, (1995).