Hubert Comon Claude Marché
Ralf Treinen (Eds.)

Constraints in Computational Logics

Theory and Applications

International Summer School, CCL'99
Gif-sur-Yvette, France, September 5-8, 1999
Revised Lectures

Springer

Volume Editors

Hubert Comon
École Normale Supérieure de Cachan, Laboratoire Spécification et Vérification
61, avenue du Président Wilson, 94234 Cachan Cedex, France
E-mail: Hubert.Comon@lsv.ens-cachan.fr

Claude Marché
Ralf Treinen
Université Paris-Sud, Laboratoire de Recherche en Informatique
Bâtiment 490, 91405 Orsay Cedex, France
E-mail: {marche/treinen}@lri.fr

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Constraints in computational logics : theory and applications ;
international summer school ; revised lectures / CCL '99.
Gif-sur-Yvette, France, September 5 - 8, 1999. Hubert Comon ... (ed.).
- Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ;
Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2002)
ISBN 3-540-41950-0

CR Subject Classification (1998): F.4.1, I.2.3, F.3.1, D.1.6, G.1.6, D.3.3

ISSN 0302-9743
ISBN 3-540-41950-0 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10782264 06/3142 5 4 3 2 1 0

Preface

CCL (*Construction of Computational Logics* and later *Constraints in Computational Logic*) is the name of an ESPRIT working group which met regularly from 1992 to 1999 (see `http://www.ps.uni-sb.de/ccl/`). It united research teams from Germany, France, Spain, and Israel, and was managed by the company COSYTEC.

In its final few years, the main emphasis of the working group was on *constraints* — techniques to solve them and combine them and applications ranging from industrial applications to logic programming and automated deduction. At the end of the working group, in fall 1999, we organized a summer school, intending to summarize the main advances achieved in the field during the previous 7 years. The present book contains the (revised) lecture notes of this school. It contains six chapters, each of which was written by some member(s) of the working group, covering the various aspects of constraints in computational logic. We intend it to be read by non specialists, though a prior knowledge in first-order logic and programming is probably necessary.

Constraints provide a declarative way of representing infinite sets of data. As we (attempt to) demonstrate in this book, they are well suited for the *combination* of different logical or programming paradigms. This is known since the 1980s for *constraint logic programming*, but has been combined with functional programming in more recent years; a chapter (by M. Rodríguez-Artalejo) is devoted to the combination of constraints, logic, and functional programming.

The use of constraints in automated deduction is more recent and has turned out to be very successful, moving the control from the meta-level to the constraints, which are now first-class objects. This allows us to keep a history of the reasons why deductions were possible, hence restricting further deductions. A chapter of this book (by H. Ganzinger and R. Nieuwenhuis) is devoted to constraints and theorem proving.

Constraints are not only a nice mathematical construction. The chapter (by H. Simonis) on industrial applications shows the important recent developments of constraint solving in real life applications, for instance scheduling, decision making, and optimization.

Combining constraints (or combining decision procedures) has emerged during the last few years as an important issue in theorem proving and verification. Constraints turn out to be an adequate formalism for combining efficient techniques on each particular domain, thus yielding algorithms for mixed domains. There is now a biannual workshop on these topics, of which the proceedings are published in the LNAI series. The chapter on Combining Constraints Solving (by F. Baader and K. Schulz) introduces the subject and surveys the results.

Before these four chapters on applications of constraint solving, the introductory chapter (by J.-P. Jouannaud and R. Treinen) provides a general introduction to constraint solving. The chapter on constraint solving on terms (by H. Comon and C. Kirchner) introduces the constraint solving techniques which are used in, e.g. applications to automated deduction.

Every chapter includes an important bibliography, to which the reader is referred for more information.

We wish to thank the reviewers of these notes, who helped us improve the quality of this volume. We also thank the European Union who supported this work for 6 years and made possible the meeting in Gif.

January 2001

Hubert Comon
Claude Marché
Ralf Treinen

Full Addresses of Contributors and Editors

Franz Baader
Rheinisch-Westfälische Technische Hochschule Aachen
Lehr- und Forschungsgebiet Theoretische Informatik
Ahornstrasse 55
D-52074 Aachen
Germany
E-mail: baader@informatik.rwth-aachen.de
Web: http://www-lti.informatik.rwth-aachen.de/ti/baader-en.html

Hubert Comon
Laboratoire Spécification et Vérification
École Normale Supérieure de Cachan
61, avenue du Président Wilson
F-94234 Cachan Cedex
France
E-mail: Hubert.Comon@lsv.ens-cachan.fr
Web: www.lsv.ens-cachan.fr/~comon

Harald Ganzinger
Max-Planck-Institut für Informatik
Programming Logics Group, Room 601
Im Stadtwald
D-66123 Saarbrücken
Germany
E-mail: hg@mpi-sb.mpg.de
Web: http://www.mpi-sb.mpg.de/~hg/

Jean-Pierre Jouannaud
Laboratoire de Recherche en Informatique
Bâtiment 490
Université Paris-Sud
F-91405 Orsay Cedex
France
E-mail: jouannau@lri.fr
Web: http://www.lri.fr/demons/jouannau

Claude Kirchner
LORIA & INRIA
615, rue du Jardin Botanique
BP-101
F-54602 Villers-lès-Nancy
France
E-mail: Claude.Kirchner@loria.fr
Web: http://www.loria.fr/~ckirchne/

Claude Marché
Laboratoire de Recherche en Informatique
Bâtiment 490
Université Paris-Sud
F-91405 Orsay Cedex
France
E-mail: marche@lri.fr
Web: http://www.lri.fr/demons/marche

Robert Nieuwenhuis
Technical University of Catalonia (UPC)
Department of Software (LSI), Building C6, Room 112
Jordi Girona 1
E-08034 Barcelona
Spain
E-mail: roberto@lsi.upc.es
Web: http://www-lsi.upc.es/~roberto/home.html

Mario Rodríguez-Artalejo
Universidad Complutense de Madrid
Dpto. de Sistemas Informáticos y Programación
Edificio Fac. Matemáticas
Av. Complutense s/n
E-28040 Madrid
Spain
E-mail: mario@eucmos.sim.ucm.es

Klaus U. Schulz
Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München
Oettingenstrasse 67
D-80538 München
Germany
E-mail: schulz@cis.uni-muenchen.de
Web: http://www.cis.uni-muenchen.de/people/schulz.html

Helmut Simonis
COSYTEC SA
4, rue Jean Rostand
F-91893 Orsay Cedex
France

Current Address:
Parc Technologies Ltd
2nd Floor, The Tower Building
11 York Road
London SE1 7NX
United Kingdom
E-mail: Helmut.Simonis@parc-technologies.com

Ralf Treinen
Laboratoire de Recherche en Informatique
Bâtiment 490
Université Paris-Sud
F-91405 Orsay Cedex
France
E-mail: treinen@lri.fr
Web: http://www.lri.fr/demons/treinen

Contents

1 Constraints and Constraint Solving: An Introduction

Jean-Pierre Jouannaud[1] and Ralf Treinen[1*]

Laboratoire de Recherche en Informatique, Université Paris-Sud, Orsay, France

1.1 Introduction

The central idea of constraints is to *compute with descriptions of data* instead of to compute with data items. Generally speaking, a constraint-based computation mechanism (in a broad sense, this includes for instance deduction calculi and grammar formalisms) can be seen as a two-tired architecture, consisting of

- A language of data descriptions, and means to compute with data descriptions. Predicate logic is, for reasons that will be explained in this lecture, the natural choice as a framework for expressing data descriptions, that is *constraints*.
- A computation formalism which operates on constraints by a well-defined set of operations. The choice of this set of operations typically is a compromise between what is desirable for the computation mechanism, and what is feasible for the constraint system.

In this introductory lecture we will try to explain basic concepts of constraint-based formalisms in an informal manner. In-depth treatment is delegated to the lectures dedicated to particular subjects and to the literature.

This lecture is organised as follows: We start with a small tour of constraint-based formalisms comprising constraint logic programming, constraints in automated deduction, constraint satisfaction problems, constrained grammars in computational linguistics, constraint-based program analysis, and constraints in model checking. In this first section, we will employ a somewhat naive view of constraint systems since we will assume that we have complete and efficient means to manipulate descriptions of data (that is, constraints). This allows to cleanly separate calculus and constraints, which can be considered as an ideal situation.

In the following sections we will see that, in reality, things are not that simple. In Section 1.3 we will investigate a particular family of constraint systems, so-called feature constraints, in some depth. It will turn out that complete procedures to compute with constraints may be too costly and that we may have to refine our calculus to accommodate for incomplete constraint

* Both authors supported by the ESPRIT working group CCL-II, ref. WG # 22457.

H. Comon, C. Marché, and R. Treinen (Eds.): CCL'99, LNCS 2002, pp. 1–46, 2001.

handling. Section 1.4 will present such a refinement for constraint logic programming. In Section 1.5, finally, we will address the question of how the basic algorithms to compute with constraints can be implemented in the same programming calculus than the one that uses these algorithms.

1.2 A First Approach to Constraint Based Calculi

1.2.1 First-Order Logic as a Language

The role of this section is to give the basics of first-order logic viewed as a language for expressing relations over *symbolic data items*. It can be skipped by those readers aware of these elementary notions.

A *term* is built from *variables*, denoted by upper-case letters X, Y, Z, etc. (following the tradition of logic programming) or by lower-case letters x, y, z, etc. (following the tradition of mathematical logic), and *function symbols* of a given *arity*, usually denoted by words of lower-case letters as f, a, *plus*. We assume a sufficient supply of function symbols of any arity. Examples of terms are a, $g(a, X)$, $f(g(X, Y), X)$ and $f(g(a, b), a)$. The number of arguments of a function symbol in a term must be equal to its given arity. Terms without variables like a or $f(g(a, b), a)$ are called *ground*, they are the data items of the logic language. The set of function symbols is called the *signature* of the language of terms.

A *substitution* is a mapping from variables to terms. Substitutions are usually denoted by Greek letters like σ, τ. Since it is usually sufficient to consider mappings that move only a finite number of variables, substitutions can be written as in $\sigma = \{X \mapsto a, Y \mapsto b\}$ and $\tau = \{X \mapsto Z\}$, where it is understood that all variables map to themselves unless mentioned otherwise. The application of a substitution to a term is written in postfix notation, and yields another term, for instance $f(g(X, Y), X)\sigma = f(g(a, b), a)$ and $f(X, Y)\tau = f(Z, Y)$. The application of a substitution to a term t yields an *instance* of t (hence a *ground instance* if the term obtained is ground). One of the basic ideas of logic is that a term denotes the set of its ground instances, hence is a description of a set of data items.

Different terms may have common ground instances, that is, the two sets of data items that they describe may have a non-empty intersection. When this is the case, the two terms are said to be *unifiable*. For an example, $f(X, g(Y))$ and $f(g(Z), Z)$ are unifiable, since they have $f(g(g(a)), g(a))$ as a common ground instance. The substitution $\tau = \{X \mapsto g(g(a)), Y \mapsto a, Z \mapsto g(a)\}$ is called a *unifier* of these two terms. Does there exist a term whose set of ground instances is the intersection of the sets of ground instances of two unifiable terms s and t? The answer is affirmative, and the *most general instance* of s and t is obtained by instantiating any of them by their *most general unifier* or *mgu*. In the above example, the mgu is $\sigma = \{X \mapsto g(g(Y)), Z \mapsto g(Y)\}$. It should be clear that any unifier τ is an instance of the mgu, by the substitution $\tau' = \{Y \mapsto a\}$ in our example. This is captured

by writing the composition of substitutions in diagrammatic order: $\tau = \sigma\tau'$. Renaming the variables of the most general instance does not change its set of ground instances, we say that the most general instance, and the most general unifier as well, are defined up to *renaming of variables.* It should be clear that we can unify n terms $t_1, \ldots, t_n$ as well. In practice, we often need to unify sequences $\bar{s}$ and $\bar{t}$ of terms, that is, find a most general unifier σ such that $s_1\sigma = t_1\sigma, \ldots, s_n\sigma = t_n\sigma$.

So far, we have considered finite terms only. In practice, as we will see later, it is often convenient or necessary to consider infinite terms as well. Examples of infinite terms are

$$g(a, g(a, g(a, \ldots g(a, \ldots))))$$
$$g(a, g(h(a), g(h(h(a)), \ldots g(h(\ldots(h(a)), \ldots)))))$$

The above discussion for finite terms remains valid for infinite *rational terms*, that is, infinite terms having a finite number of subterms. In our example of infinite terms, the first is regular while the second is not. An alternative characterisation of regular terms is that they can be recognised by finite tree automata. Regular terms can therefore be described finitely, in many ways, in particular by graphs whose nodes are labelled by the symbols of the signature. Apart from its complexity, unification of infinite regular terms enjoys similar properties as unification of finite terms.

An *atom* is built from terms and *predicate symbols* of a given arity, usually also denoted by words of lower-case letters. There is no risk of confusion with function symbols, since terms and atoms are of two different kinds. We assume a sufficient supply of predicate symbols of any arity. Examples are *connect(X,Y,join(a,b))* or *append(nil,cons(a,nil),cons(a,nil))*, the latter is called *closed* since it contains no variable.

Formulae are built from atoms by using the binary logical connectives $\wedge$, $\vee$ and $\Rightarrow$, the unary connective $\neg$, the logical constants $True$ and $False$, the existential quantifier $\exists$ and the universal quantifier $\forall$. Several consecutive universal or existential quantifications are grouped together. An example of a formula is $\forall XY \exists Z\ plus(X, Z, succ(Y))$. *Clauses* are particular formulae of the form $\forall \overline{X}\ B_1 \vee \ldots \vee B_n \vee \neg A_1 \ldots \vee \neg A_m$, where $m \geq 0, n \geq 0$, and $\overline{X}$ denotes the sequence of variables occurring in the formula. The atoms B_i are said to be *positive*, and the A_j *negative*. The universal quantification $\forall \overline{X}$ will usually be implicit, and parentheses are often omitted. The clause may equivalently be written $(A_1 \wedge \ldots \wedge A_m) \Rightarrow (B_1 \vee \ldots \vee B_n)$, using conjunctions and implication, or simply $B_1, \ldots, B_m \leftarrow A_1, \ldots, B_n$, a notation inherited from Logic Programming. Clauses with at most one positive atom are called *Horn clauses*, the positive atom is called its *head*, and the set of negative ones form its *body*. A Horn clause without body is called a *fact*, and one without head a *query*.

To know more: More on unification of terms and automata techniques is to be found in the chapter *Constraint Solving on Terms.* See also [JK91] for

a modern approach to unification and [CDG+99] on automata techniques. The basic logic notations can be found in many text book covering logical methods, for instance [BN98].

1.2.2 From Logic Programming to Constrained Logic Programming

Historically, the need of constraints arose from *Logic Programming* (LP), one of the classical formalisms following the paradigm of *declarative programming.* Let us recall the basics of logic programming before we show how constraints make LP an even more natural and efficient declarative language. A *program* is a sequence of *definite* Horn clauses, that is, Horn clauses with a non-empty head. The declarative semantics of a logic program P is the least set M_P of ground atoms such that if $B \leftarrow A_1, \ldots, A_n$ is a ground instance of a program clause of P and if $A_1, \ldots, A_n \in M_P$ then $B \in M_P$. Such a least set does always exist for Horn clauses (but not for arbitrary clauses).

Here is an example of logic program:

```
arc(a,b).
arc(b,c).
arc(c,d).
arc(c,e).
path(X,X).
path(X,Y) ← arc(X,Z),path(Z,Y).
```

The idea of this program is that ground terms a, b, c, etc. are nodes of a graph, *arc* defines the edges of the graph, and $path(X, Y)$ whether there is a directed path from X to Y.

While functional or imperative programs execute when provided with input data, logic programs execute when provided with *queries*, that is, a conjunction of atoms, also called a *goal.* Asking whether there is an arc from c to e is done by the query $\neg arc(c, e)$, which we could also write $\longleftarrow arc(c, e)$. Knowing that it is a query, we will simply write it $arc(c, e)$. The answer to this query should be affirmative, and this can simply be found by searching the program facts. In contrast, the answer to the query $arc(c, X)$ should be all possible nodes X to which there is an arc originating in c. Here, we need to non-deterministically unify the query with the program facts. Finally, the answer to the query $path(X, e)$ should be all nodes X from which there is a path ending in e. Here, we need to reflect the declarative semantics of our program in the search mechanism by transforming the query until we fall into the previous case. This is done by the following rule called *resolution*:

(Resolution) $\dfrac{P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots P_n(\bar{t}_n)}{Q_1(\bar{r}_1\sigma), \ldots, Q_m(\bar{r}_m\sigma), P_2(\bar{t}_2\sigma), \ldots, P_n(\bar{t}_n\sigma)}$
where $P_1(\bar{s}) \leftarrow Q_1(\bar{r}_1), \ldots, Q_m(\bar{r}_m)$ is a program clause not sharing any variable with the query $P_1(\bar{t}_1), \ldots, P_n(\bar{t}_n)$ and σ is the most general unifier of $\bar{s}$ and $\bar{t}_1$.

This rule contains two fundamentally different non-determinisms: the choice of the atom to which resolution is applied is don't-care (it does not matter for the completeness of the search procedure which atom we choose to resolve on), while the choice of the program clause is don't-know (we have to try all possible clauses). This non-determinism is dealt with externally: choosing an atom in the query, we have in principle to execute all possible sequences of resolutions steps. Along every execution sequence we construct the sequence of substitutions computed so far. If such a computation sequence terminates in an empty query, then we say that the computation *succeeds* and that the computed substitution is an *answer substitution*, otherwise it *fails*.

In pure LP, all data items are represented as ground terms, and unification is the fundamental method of combining descriptions of data items. If data types like natural numbers are to be used in LP they have to be encoded as terms, and operations on numbers have to be expressed as LP predicates. Let us, for example, extend our example of paths in graphs to weighted graphs:

```
plus(0,X,X).
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
arc(a,b,s(0)).
arc(b,c,s(s(0))).
arc(c,d,s(0)).
arc(c,e,s(0)).
path(X,X,0).
path(X,Y,L) ← arc(X,Z,M),path(Z,Y,N),plus(M,N,L).
```

With this program, the goal $path(a, c, L)$ will succeed with answer substitution $\{L \mapsto s(s(s(0)))\}$. However, computation of the goal $plus(X, s(0), X)$ will not terminate. The problem is that, although searching solutions by enumerating all possible computation paths is complete in the limit, this gives us in general not a complete method for detecting the absence of solutions (not to speak of its terrible inefficiency). What we would like here is to have numbers as a primitive data type in LP.

Another problem with pure LP is that, even with terms as data structures, we might want to have more expressiveness for describing terms, and be able, for instance, to formulate the query "is there a path of weight 10 starting in a and ending in a node different from e"? Our approach to use terms with variables as descriptions of ground terms is obviously not sufficient here.

Constraint Logic Programming (CLP) offers a solution to these deficiencies of LP. The basic idea is to disentangle the resolution rule, by separating the process of replacing an atom of the query by the body of a program clause from the other process of unifying the data descriptions. For this, we need to consider a new syntactic entity, namely *equations* between terms. These equalities must be distinguished from possibly already existing equality atoms, since their use will be quite different. In addition, this will allow us to restrict our syntax for atoms to predicate symbols applied to *different* variables, the value of these variables being recorded via the equalities. A Horn clause is now of the form

$$B \leftarrow A_1, \ldots, A_n, c$$

where B and the A_i are atoms and c is a conjunction of equations usually written again as a set. Our example of weighted graphs could now look like this:

```
plus(X,Y,Z) ← X=0,  Y=Z.
plus(X,Y,Z) ← plus(X',Y,Z'), X=s(X'), Z=s(Z').
arc(X,Y,Z)  ← X=a, Y=b, Z=s(0).
arc(X,Y,Z)  ← X=b, Y=c, Z=s(s(0)).
arc(X,Y,Z)  ← X=c, Y=d, Z=s(0).
arc(X,Y,Z)  ← X=c, Y=e, Z=s(0).
path(X,Y,L) ← X=Y, L=0.
path(X,Y,L) ← arc(X,Z,M), path(Z,Y,N), plus(M,N,L).
```

In this new context, queries are called *configurations*. A configuration consists of a list of atoms *followed by* a list of equations. We can rewrite our resolution rule as:

(Resolution-2) $$\frac{P_1(\bar{X}_1), P_2(\bar{X}_2), \ldots, P_n(\bar{X}_n), c}{Q_1(\bar{Y}_1), \ldots, Q_m(\bar{Y}_m), P_2(\bar{X}_2), \ldots, P_n(\bar{X}_n), c, d}$$

where $P_1(\bar{X}_1) \leftarrow Q_1(\bar{Y}_1), \ldots, Q_m(\bar{Y}_m), d$ is a program clause not sharing any variable with the configuration $P_1(\bar{X}_1), \ldots, P_n(\bar{X}_n), c$ except $\bar{X}_1$, and $c \wedge d$ is satisfiable.

Unification has disappeared from the resolution rule: it is replaced by constructing the conjunction of the two equation systems and checking their satisfiability. Note that in the constrained resolution rule the parameters $\bar{X}_i$ of the predicate P_i are required to be the same lists of variables in the query and in the program clause, this will have to be achieved by renaming the variables in the program clause when necessary.

Semantically, an equation $X = s$ must be understood as the set of values (for the variable X) which are the ground instances of the term s. When making a conjunction $X = s \wedge X = t$, we intersect these two sets of values, and therefore, the set of solutions is the same as that of $X = most\text{-}general\text{-}$

instance(s,t), provided s and t are unifiable. This is why we need to check for satisfiability in our new rule. This satisfiability test will of course be delegated to the old unification algorithm, which is seen here as an "equation solver" which is put aside. Furthermore, we could allow more freedom in the rule by writing in the conclusion of the rule any equations system that is equivalent to $c \wedge d$. This gives the freedom of whether to leave the constraint $c \wedge d$ as it is, or to replace it by some simpler "solved form" (see the chapter *Constraint Solving on Terms* for possible definitions of solved forms).

Execution of our program is now essentially the same as before, we just have separated matters. An important observation is that we have only used the following properties of descriptions of terms:

- we can intersect their denotation (by building the conjunction of the associated equalities)
- we can decide the emptiness of the intersection of their denotations (by unification).

The crucial abstraction step is now: We obtain CLP when we replace equation systems of terms by any system of description of data with the same two properties as above. For example, we could replace the Presburger language for integers used in our last example by the usual integer arithmetic:

```
arc(X,Y,Z)  ← X=a, Y=b, Z=1.
arc(X,Y,Z)  ← X=b, Y=c, Z=2.
arc(X,Y,Z)  ← X=c, Y=d, Z=2.
arc(X,Y,Z)  ← X=c, Y=e, Z=2.
path(X,Y,L) ← X=Y, L=0.
path(X,Y,L) ← arc(X,Z,M), path(Z,Y,N), L = M + N.
```

To know more: An introduction to Logic Programming from a theoretic point of view is to be found in [Apt90].

1.2.3 Constraint Systems (First Version)

We are now ready for a first, naive version of constraint systems. First, we use predicate logic for syntax and semantics of constraints since it provides the following features which are important for most of the constraint-based formalisms:

1. first-order variables to denote values, these variables will be shared with the formalism that uses the constraints (for instance a programming language),
2. conjunction as the primary means to combine constraints, since conjunction of formulas corresponds exactly to the intersection of their respective denotation,

3. existential quantification to express locality of variables. It is existential, in contrast to universal, quantification that matters here since the constraint-based mechanisms that we consider are based on *satisfiability* of descriptions.

Hence, a first definition of a *constraint system* (we will refine this definition in Section 1.4.1) could be as follows:
A constraint system C consists of

- A language L_C of first-order logic, that is a collection of function symbols and of predicate symbols together with their respective arities. The atomic formulae built over this language are called the *atomic constraints* of C.
- a subset of the set of first-order formulae of L_C, called the set of *constraints* of C, containing all atomic constraints, and closed under conjunction, existential quantification and renaming of bound variables. Very often, the set of constraints is the minimal set containing the atomic constraints and with the above properties, in which case we call it the set of *basic* constraints of L_C.
- a first-order structure A_C of the language L_C,
- an algorithm to decide whether a constraint is satisfiable in A_C or not.

Some examples of constraint systems are:

1. *Herbrand*: This is the constraint system used in LP when seen as instance of the CLP scheme. Its language is given by an infinite supply of function symbols and equality as the only predicate. The set of constraints is the set of basic constraints whose atomic constraints are the equality atoms, the structure is the structure of ground terms, where function symbols are just constructors of ground terms and equality is completely syntactic, and the algorithm for testing satisfiability is the unification algorithm (usually attributed to Herbrand).
2. *Herbrand over rational terms*: This is the constraint system introduced by Colmerauer in PROLOG II. It is the same as previously, but the structure is the set of ground atoms built from rational terms. The algorithm for testing satisfiability is due to Huet.
3. *Herbrand with inequations*: This is another extension of Herbrand where constraints are existentially quantified systems of equations and inequations. Decidability of satisfiability of constraints has been shown by Alain Colmerauer. This constraint system was the basis of PROLOG II.
4. *Presburger arithmetic*: The language of Presburger arithmetic consists of a constant 0, a unary function s, and a binary operator $+$. Constraints are the basic constraints, interpreted in the usual way over natural numbers. There is a number of standard methods to decide satisfiability of these constraints. The set of constraints can be extended to the full set of first-order formulas. There are many ways to show decidability of the