Constraints, , 1–26 (1997)

Submission to CONSTRAINTS

A Methodological View of Constraint Solving

HUBERT COMON hubert.comon@lsv.ens-cachan.fr
LSV, Ecole Normale Supérieure de Cachan, 61 Ave. Président Wilson, 94235 Cachan cedex, France

MEHMET DINCBAS dincbas@cosytec.fr
COSYTEC, Parc Club Orsay Université, 4 Rue Jean Rostand, 91893 Orsay Cedex, France

JEAN-PIERRE JOUANNAUD jouannaud@lri.lri.fr
CNRS and LRI, Bat. 490, Université de Paris Sud, 91405 ORSAY Cedex, France

CLAUDE KIRCHNER claude.kirchner@loria.fr
INRIA Lorraine & CRIN, 615 rue du jardin botanique, BP 101, 54602 Villers-lès-Nancy Cedex, France. `http://www.loria.fr/~ckirchne/`

Received June 1997

Editor:

Abstract. Constraints are an effective tool to define sets of data by means of logical formulae. Our goal here is to survey the notion of constraint system and to give examples of constraint systems operating on various domains, such as natural, rational or real numbers, finite domains, and term domains. We classify the different methods used for solving constraints, syntactic methods based on transformations, semantic methods based on adequate representations of constraints, hybrid methods combining transformations and enumerations. The concepts and methods are illustrated via examples. We also discuss applications of constraints to various fields, such as programming, operations research, and theorem proving.

Keywords: Constraint solving, Constraint satisfiability, Unification, Rule based solving.

1. Introduction

Constraints allow to define sets of data by means of logical formulae. For example, the set of even numbers y given *in extension* as the set $\{0, 2, 4, \ldots\}$, can be described *in comprehension* by the formula $\{y \in \mathbb{N} \ : \ \exists x \in \mathbb{N}\ y = 2 * x\}$ stating that there exists a natural number x such that y is twice x. In this formula, the subformula $\exists x\ y = 2 * x$ is the *constraint*, and $\mathbb{N}$ is its *domain of interpretation*. Constraints have become very successful in computer science and operation research. There are several reasons for this, some of which are listed below.

A convenient notation. Constraints provide a compact, simple, and familiar representation of possibly infinite sets, which is a first reason for their success.

Expressivity. A second reason proceeds from a more pragmatic argument: constraints allow to express and manipulate *large* sets of data that could not be reasonably handled if given in extension.

Describing the infinite set of even numbers in extension means to give a procedure than can enumerate this set by outputing the value $2n$ upon input n. Defining the set of multiples of 3 can be done in the same manner. The intersection of the two sets can now be computed by enumerating them both alternatively, and outputing the common values: $6, 12, 18, \ldots$. There is a deadly risk, of course, when the intersection is empty. In contrast to this *procedural* style, constraints are *declarative*: they allow to *state* properties independently of an actual computation mechanism, called *solving*. In the above example, the set of triples is specified as $\exists z\ y = 3 * z$ with $\mathbb{N}$ for domain of interpretation. The intersection of the two sets can be simply expressed by *accumulating* both constraints, resulting in $\exists x\ y = 2 * x\ \&\ \exists z\ y = 3 * z$. Constraints are indeed targetted at specifying set-theoretic properties.

Efficiency. Declarative languages are often considered as inefficient. A third important reason why constraints are so useful, is that they reconcile declarativeness with efficiency. A constraint is solved in a given domain of interpretation, hence very efficient algorithms can be chosen for this purpose and can in practice be fetched from a library. Other declarative languages would use instead a general purpose solving algorithm called *search*. For example, a standard Prolog implementation would solve a system of linear equations over the rationals by using SLD resolution, while a language allowing for rational constraints can call Gaussian elimination, and therefore be much more efficient.

Easy combination. There is yet a fourth reason for the success of constraints: they can be very easily *combined* with other computation models. For example, combining logic programming and constraints is achieved by replacing the unification component of a logic programming language by constraint solving algorithms. In this combination, the logic programming computation produces new constraints which are consumed by the constraint solving algorithms, which in turn produces new inputs for the logic programming computation. The ability to efficiently compute with sets of constraints which evolve along a computation is a last major property of constraints called *incrementality*.

Constraints play a key role since the work of Alan Colmerauer [6], their abstract formulation by Joxan Jaffar and Jean-Louis Lassez [26], and the introduction of finite domain constraints in CHIP [16].

So far, we have not explained a major aspect of constraints, directly related to the title of this paper. There are (in general infinitely) many ways to represent a set of data by means of constraints. For example, the set of even numbers could have been defined as the set of non-odd natural numbers z, that is by the constraint $[\forall x\ (\exists y\ x = 2 * y + 1) \Rightarrow z \neq x]$. This is of course a more complicated equivalent formula, but what does *more complicated* mean? There is no definite answer to this question, unless the formula we are interested in defines the empty set. Indeed, it is a major problem to avoid useless computations by efficiently checking constraints for emptiness.

To summarize, a constraint system comes in three parts:

1. The syntax in which the logical formulae are expressed. In most cases, it is a first-order language, or a fragment thereof closed under the conjunction operator &. Very common are the *existential* fragment, which is free of universal quantifier, and the *positive* fragment, for which negation and disjunction are not allowed. Constraints without logical connectives are called *atomic*.

   A typical example consists in the conjunctions of linear equations over rational numbers, such as $y = \frac{4}{3}x \ \& \ x + y = 63$.

2. The structure in which the formulae are interpreted, composed of a domain of values on one hand, and, on the other hand, of operations and relations interpreting all symbols in the syntax.

   In our example, the domain of values is the set of rational numbers, with the usual operations.

   A *solution* of a constraint assigns its free variables to values of the domain in such a way that the formula evaluates to `True`. In the above example, there is only one solution: $x := 27, y := 36$.

3. A *constraint solving* algorithm takes as input a constraint C and returns as output a particular representation of the set of all its solutions. As an important particular case, this algorithm allows to decide whether C has at least one solution, this is called *constraint satisfaction*. A more general algorithm is often needed, called *constraint entailment*, which decides whether the set of solutions of a constraint C is included into the set of solutions of a constraint C'.

   Gaussian elimination is one algorithm used to solve linear equations over rational numbers.

We describe later in detail constraints on finite domains, which pop up in many application areas. Such constraints are NP-hard, which makes it impossible to deal with large application problems. As a consequence, practice sometimes favors *incomplete* algorithms, which may sometimes fail to detect that a given constraint has no solution. On the other hand, a constraint solving algorithm must always be *sound*: it should not pretend that there is no solution when there is some.

So far, all our examples deal with numbers. Indeed, constraints over numbers or Booleans have many industrial applications as operations research and hardware verification. But, there are also applications to computer science, for which the domain is itself made of expressions, called *terms* in this context. A typical example is the problem of solving equations over terms, called *unification*, which is at the heart of logic programming. We call *symbolic*, constraints over terms.

Symbolic constraints have been used for years in computer science in order to:

- represent sets of formulae. A *constrained formula* is a pair $(\phi \mid C)$ which stands for the set of instances of ϕ by the solutions of C. Consider arithmetic expressions built over the addition $+$ and the multiplication $*$ as they are found in programming languages. The subset of these expressions which are sums of two different expressions can be expressed as the formula $\exists xy\ (z = x + y \mid x \neq y)$. Here, $x \neq y$ means that x and y stand for two syntactically different expressions. This example shows the expressive power of constraints, since the above set cannot be defined by finite automata operating on trees, as the ones used later in this article.

- avoid useless processing of identical subterms by sharing them. This is used in most interpreters or compilers under the name of an environment binding the variables. For example, $(x + x \mid x = t)$, where t may be a very large expression.
- represent infinite terms, the structure used by most implementations of logic programming languages. For example, the infinite list of zeros can be represented by the constraint $x = cons(0, x)$, interpreted in the domain of lists of natural numbers.
- express search strategies, as in automated deduction, allowing to prune further the search space.

The need for more complex *combined constraints* involving several domains of interpretation arises in many situations. For example, integers are often used to describe parameterized families of terms, by indicating the repetition of a given subexpression. Constraints operating on such terms are then made of two components, an integer constraint, and a symbolic constraint. Another example, that we will work out in more detail in the sequel, is provided by bibliographic data bases. Querying the data base for all titles containing a given word can be expressed as an entailment problem over a language combining feature constraints for expressing record definitions, with word constraints for expressing the search of a given word.

Our goal in this paper is to describe the different methods on which constraint solving algorithms are based. We distinguish among: (complete) *syntactic methods*, which do not commit to any particular representation of the problem domain or the set of solutions; (complete) *semantic methods*, which rely on a particular representation of the problem domain as well as of the set of solutions; *hybrid methods* which combine (incomplete) syntactic or semantic steps together with non-deterministic steps based on a partial enumeration of the set of solutions. These three methods are described respectively in sections 2, 3, 4. Applications areas are then considered in section 5.

2. Syntactic Transformations

2.1. *The general setting*

Syntactic methods are based on a very simple principle: to repeatedly transform a constraint into an equivalent one until a so-called *solved form* is obtained. Most computations can be seen as such a rewriting process. However, we insist here on the fact that our transformation rules are applied on the formulas themselves, as opposed to some coding of them: the rules simply match the (sub)formulas against some given patterns and replace them with simpler formulas. We will see some techniques which do not fall in this category in the next two sections.

Assume that I have twice the age you had when I had the age you have. Since I'm not 48, and you are not 38, how old are we? This can be expressed by the following constraint over rational numbers:

$$\exists t \quad x = 2 * (y - t) \quad \& \quad x - t = y \quad \& \quad x \neq 48 \quad \& \quad y \neq 38$$

The transformation proceeds as you expect, by choosing a variable, say x and replacing it within the other equations, yielding the new *equivalent* system, that is having the same set of solutions:

$$\exists t \quad x = 2 * (y - t) \quad \& \quad 2 * (y - t) - t = y \quad \& \quad 2 * (y - t) \neq 48 \quad \& \quad y \neq 38$$

The system is then simplified according to elementary rational number theory, resulting in:

$$\exists t \quad x = 2 * (y - t) \quad \& \quad t = \frac{y}{3} \quad \& \quad 2 * y - 2 * t \neq 48 \quad \& \quad y \neq 38$$

We now proceed by eliminating t together with its existential quantifier:

$$x = \frac{4}{3} y \quad \& \quad y \neq 36 \quad \& \quad y \neq 38$$

which is in solved form. Hence, there are solutions, such as, e.g. $x = 56$ and $y = 42$.

Figure 1. What is their age?

Figure 1 illustrates these concepts by developping an example of solving linear constraints over rationals. This technique is attributed to Gauss, the celebrated German mathematician of the 19th century, but was actually in use before that. A different problem is obtained by using inequalities instead of equalities. This problem can be reduced to the first by adding the so-called *slack variables* which express the difference between the two sides of the equation. For example, the constraint $x \leq y$ is equivalent to the constraint $\exists z \; x + z = y \; \& \; z \geq 0$. This is of course another formula transformation. With linear programming comes a more difficult problem, in which a linear function must be optimized with respect to the set of solutions of a constraint of the above form. An example is described in figure 2, with the general principles of Dantzig's *simplex* algorithm [14]. Although exponential, it is often prefered to more recent polynomial algorithms such as the "interior point methods" [32] for two main reasons: it performs quasi linearly in the average, and has an efficient incremental version.

Our examples show the ingredients of the transformation method. The constraint is a logical formula. Each rule transforms a constraint into an equivalent one, the

Vincent Mongeard-Mugneret, wine maker in Burgundy, just bought 20 acres of good soil at Vosne Romanée, planted with red pinot. He has two options, which cannot be changed once they are chosen. By bringing manure from his friend's farm, his investment per acre-year will be equal to \$1000, and will rise up to \$3000 if he uses chemical fertilizer. One acre of the first kind will produce 1000 bottles a year sold \$25 dollars each, while the second will produce 50% more sold \$20 each. His investment for the next 5 years is limited to \$200000. How many acres should be fertilized with chemicals, and how many with manure in order to maximize the profit?

Let us take x_1 for the number of acres cultivated with manure, x_2 for the number of acres cultivated with chemicals, z_1 (number of non-cultivated acres) and z_2 (unused investment) are the slack variables. All variables are positive.

$$\left\{\begin{array}{rcl} x_1 + x_2 + z_1 & = & 20 \\ 5000x_1 + 15000x_2 + z_2 & = & 200000 \\ \multicolumn{3}{l}{\text{Maximize}\{125000x_1 + 150000x_2 - 5000x_1 - 15000x_2\}} \end{array}\right.$$

The process repeats the following steps: (i) choose one distinct variable per equation, (ii) solve the system with respect to the choosen variables, (iii) replace the left-hand side variables in the profit function. The choice of the variables at step (i) must ensure that the right-hand side constants in the equations obtained at step (ii) are all positive, and the constant in the profit function obtained at step (iii) has not decreased.

$$\left\{\begin{array}{rcl} z_1 & = & 20 - x_1 - x_2 \\ z_2 & = & 200000 - 5000x_1 - 15000x_2 \\ \multicolumn{3}{l}{\text{Maximize}\{15000(8x_1 + 9x_2)\}} \end{array}\right.$$

Pivoting on x_1 and x_2 yields

$$\left\{\begin{array}{rcl} x_1 & = & 10 - \frac{3}{2}z_1 + \frac{z_2}{10000} \\ x_2 & = & 10 - \frac{z_2}{10000} + \frac{z_1}{2} \\ \multicolumn{3}{l}{\text{Maximize}\{15000(170 - \frac{15}{2}z_1 - \frac{z_2}{10000})\}} \end{array}\right.$$

The profit function is clearly maximized by taking $z_1 = z_2 = 0$, since their coefficients are negative. This yields a profit of \$255000[1].

Figure 2. How good is your *Vosne Romanée?*

equivalence proof relying on some algebraic property of the constraint domain. Then each rule comes with some conditions on its application, which is supposed to ensure the termination of the overall transformation process. The rules are not necessarily deterministic: it may happen that several rules can be applied to the same constraint, or it may happen that two distinct instances of the same rule can be applied at the same time. For example, solving systems of linear equations, at each step we may choose to eliminate any (unsolved) variable, while keeping the termination property. This non-determinacy is quite different from e.g. non-determinism in PRLOG programs: this is a "don't care" non-determinism. We can commit to any choice, without taking care of other possible transformations. Then, an actual implementation has to consider in addition a *strategy* which restores the determinism. For instance, in the case of linear equation solving, the strategy will specify an ordering on the pivoting variables. This distinction between the strategy and the rules themselves is important as it allows to reuse the same set of rules (and the same termination proof) together with distinct strategies, which yield efficient implementations depending on the particular properties of the constraints. For example, the ordering on variables to be eliminated may depend on the form of the system.

Finally, a particular set of constraints called *solved forms* has to be specified. Such solved forms are supposed to have some additional properties. For example, in the case of a system of linear equations, a solved form is a system of equations such that every left member is a variable not occurring anywhere else. Strategies should be designed in such a way that they are *complete*, i.e. every irreducible constraint should be in solved form.

Formalizing this approach, the following steps are necessary:

1. Choose a set of *solved forms*, that is a set of formulae that have no solution if and only if they are syntactically equal to the formula $\bot$, standing for the empty set.

2. Design a set of *transformation rules* and show that it has the following properties:

   (A) *Correctness*: applying a rule from the set to an arbitrary input constraint results in a new constraint which has the same set of solutions. Proving correctness is usually manageable since it breaks down into elementary correctness proof for each transformation rule.

   (B) *Termination*: given an arbitrary constraint, every sequence of transformation originating from it results in an irreducible constraint after a finite number of steps. This property may be hard to prove.

3. Design a strategy which yields a deterministic set of rules and having the following property:

   (C) *Completeness*: there are enough rules, that is, for each constraint which is not a solved form, at least one rule applies. Therefore, the algorithm never fails to detect that a given constraint has no solution. This property is usually simple to prove and easy to enforce.

As part of its conceptual simplicity, an important advantage of this approach is to provide a systematic guide for constraints solving. When it applies, this method yields an algorithm which is inherently *incremental*, since new constraints can always be added (by using conjunction) to a solved form, yielding a new constraint equivalent to the conjunction of the starting constraint and the new one.

2.2. Unification and its extensions

The approach described in the previous section was indeed successfully used for a variety of problems, starting with J. Herbrand [24] who was interested in solving equations over terms, for automated deduction purposes. This now classical abstract formulation of the well known unification algorithm was rediscovered in the late 70's by Martelli and Montanari [37], who formulated it from a more operational point of view. The presentation of the unification transformation rules given in figure 3, is inspired from [6] and [28].

This general approach has been applied in a systematic way since the beginning of the eighties in two different directions which both show the power of the approach: to reformulate existing constraint solving algorithm; to generalize the unification algorithm in a number of directions. Let us mention now some of these generalizations.

Equational unification consists in solving equations over terms when the function symbols considered satisfy certain equational axioms. The problem here is that using the **Decompose** rule is no longer complete. For example, if the head symbol f in the rule is commutative (i.e. the order of its two arguments is irrelevant), then the equation $f(s_1, s_2) = f(t_1, t_2)$ should be splited in the two possibilities: either $s_1 = t_1$ & $s_2 = t_2$, or $s_1 = t_2$ & $s_2 = t_1$. Although it is still correct in this particular case, the **Check** rule is no more correct in general. Many investigations in equational unification were motivated by the practical need of building equational knowledge in the resolution inference rule of theorem provers. This is the case, for example, of commutativity and associativity which occurs in many useful algebraic structures, or distributivity (unification modulo distributivity has been shown recently to be decidable [45]). But automated deduction is not the only application area where these techniques are eventually used. Let us mention among many: — a theory arising in the context of library search, related to the axiomatization of cartesian closed categories; — a theory originating from type inference for records in object oriented functionnal languages; — the theory of Boolean rings, which study was motivated by harware verification problems.

Roughly speaking, unification constraints are formulae where quantifier and negation do not appear. Quantifiers occur of course naturally in the expression of various problems, hence more general constraints arise in many applications. Arbitrary first-order constraints built upon the equality predicate interpreted over terms are called *equational constraints* [7]. For example, a method for building

The unification transformation rules are parameterized by the vocabulary used for building expressions, called terms. We use f and g for arbitrary function symbols, x and y for variables, and s and t for arbitrary terms. Conjunctions of equations between terms are called unification constraints. Solved forms are chosen to be assignements of terms to variables, that is constraints of the form $x_1 = t_1 \;\&\; \ldots \;\&\; x_n = t_n$ in which any of the variables x_i occurs exactly once (see [28] for other choices). To obtain a solved form from an arbitrary unification constraint, several transformation rules are needed, among which two are most important: decomposition simplifies an equation whose left and right hand side terms are rooted by the same function symbol; elimination *propagates* when necessary the value of a variable to the rest of the unification constraint. The whole set of transformation rules for unification is the following:

Delete

$$s = s \;\&\; P \quad \rightarrow \quad P$$

Decompose

$$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \;\&\; P \quad \rightarrow \quad s_1 = t_1 \;\&\; \ldots \;\&\; s_n = t_n \;\&\; P$$

Conflict

$$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_n) \;\&\; P \quad \rightarrow \quad \perp \qquad \textbf{if} \quad f \neq g$$

Check

$$x = s \;\&\; P \quad \rightarrow \quad \perp \qquad \textbf{if} \quad x \text{ occurs in the non-variable term } s$$

Eliminate

$$x = s \;\&\; P \quad \rightarrow \quad x = s \;\&\; P \text{ in which } x \text{ is replaced by } s \qquad \textbf{if} \quad x \text{ does not occur in } s \text{ and occurs in } P$$

Swap

$$s = x \;\&\; P \quad \rightarrow \quad x = s \;\&\; P \qquad \textbf{if} \quad x \text{ is a variable, } s \text{ is not a variable}$$

In the above rules, the conjunction & is supposed to be associative and commutative, allowing us to single out any equation from the constraint. As an example of use, the unification constraint $f(x, g(a, x)) = f(h(a), g(a, y))$ can be transformed as follows:

$$\begin{array}{ll}
 & f(x, g(a, x)) = f(h(y), g(a, h(z))) \\
\rightarrow_{Decompose} & x = h(y) \;\&\; g(a, x) = g(a, h(z)) \\
\rightarrow_{Decompose} & x = h(y) \;\&\; a = a \;\&\; x = h(z) \\
\rightarrow_{Eliminate} & x = h(y) \;\&\; h(y) = h(z) \\
\rightarrow_{Decompose} & x = h(y) \;\&\; y = z \\
\rightarrow_{Eliminate} & x = h(z) \;\&\; y = z
\end{array}$$

which is in solved form. The associated assignment $\{x = h(z), y = z\}$ is called a most general unifier of the starting unification constraint. Any other assigment solution of the constraint can indeed be obtained from a most general unifier by an appropriate *specialization*.

Figure 3. Unification constraints

counter examples in theorem proving uses equational constraints with quantifiers and negations [4]. Jouannaud and Kounalis method for inductive theorem proving [30] uses equational constraints with universal quantifiers and negation (see [7]). Equational constraints also appear in learning from examples and counter-examples as described by Lassez and Marriott [35]. Equational constraints can be reduced to a solved form, as shown independenlty by Comon and Maher [11, 36]. These solved forms are purely existential, hence *quantifier elimination* rules become necessary. Although there is still a limited amount of negations in these solved forms, there are also *negation elimination* rules. And indeed, it has been an important problem in this area, whether negations could always be eliminated when an equivalent negation free solved form existed. A positive answer has been given by Tajine [48] to this important problem which has applications to compilation of pattern-matching definitions, and also to inductive inference from both examples and counter-examples. Of course, some operators may again satisfy certain axioms, and again associativity and commutativity are important for applications. Unfortunately, it is undecidable to know whether an arbitrary equational constraint has solutions in this particular case [52], and this is true as well for most algebraic structures.

2.3. Some other constraint systems

There are other important predicates in practice. Term orderings, for example, surface naturally in mechanizing deduction since they allow reducing the search space by eliminating those inferences which do not satisfy certain ordering conditions (see [9, 31, 41, 12] among others for term ordering constraint solving). Membership predicates are important in typed languages where they are interpreted as set membership (see e.g. [10] for constraint solving). Set predicates have been used for defining and computing partial interpretations of logic programs (see [23]. Again, the problem becomes more difficult, in general undecidable, when some operators additionally satisfy certain equational axioms.

First-order terms (that we have used up to now) lack the possibility to bind (possibly functional) variables as in higher-order logic and functional programming. Unfortunately, even unification constraints become undecidable in this setting. However, they are semi-decidable, and it turns out that Huet's semi decision procedure [25] is used in most proof development systems. An important decidable subcase, called pattern unification [39], is used in the language λ-PROLOG, a powerful extension of ordinary PROLOG.

Function symbols have a fixed number of arguments. It is sometimes convenient to represent a term as a record, by using numeric keys for retrieving subterms, as in $f(1 \Rightarrow s_1, \ldots, n \Rightarrow s_n)$. Such expressions have been used for quite some time in natural language processing [44], and more generally in knowledge representation, under the name of *features*. They have also been used together with subtyping mechanims in the area of programming languages and data bases to model inheritance [?]. The figure 4 describes a possible syntax for feature constraints, and suggests how the unification rules may be adapted to this case. The area of feature constraints has been very active in the last 10 years, and resulted in a complete constraint solver for first-order features on one hand [3, 53], and in many prototype implementations of feature-based languages on the other hand [2, 46]. We refer to [53] for more information on feature constraints.

Finally let us note that most of the efforts in symbolic constraint solving are surveyed in [28, 7] See also [29, 42] for more recent works.

3. Semantic Methods

Semantic methods, as opposed to syntactic ones, do not operate directly on the constraint syntax. Instead, they use another representation of the constraint and of its solutions, so that constraint solving relies then on the use of a specific data structure.

3.1. Automata techniques

A typical example is the use of automata. The idea, which goes back to Büchi in the early sixties, consists in associating with each formula defining a constraint an automaton recognizing the solutions of the constraint (see e.g. [49, 8]). An automaton is a very simple *finite transition system* which reads a string of input symbols sequentially, and upon each reading, moves from one state to another depending on its current state. A word is recognized if the automaton ends up in a distinguished final state.

Once an automaton has been associated with each atomic constraint, the construction proceeds by replacing logical operations on constraints by set operations on automata: conjunction yields intersection, disjunction yields union, negation yields complement, existential quantification yields projection. This is so because logical operations on constraints correspond to set operations on their solution sets. For instance, the solutions of a conjunction of two constraints C_1 and C_2 are in the intersection set of the solutions of C_1 and C_2 respectively. Since automata are closed under set operations, the automaton associated with C_1 & C_2 is $A_1 \cap A_2$ if A_1 and A_2 accept the solutions of C_1 and C_2 respectively. Projection has to be explained a little bit further: the automata have to recognize *tuples* with as many elements as the number of free variables in the constraint. Projection consists in forgetting one of the components of the tuple, while keeping the same control in the automaton.

Once an automaton A is associated with the constraint C, the satisfaction problem for C is equivalent to the emptiness decision problem for A that is: C has at least a solution iff A accepts at least one word (or tree or graph). The automaton A is then a representation of the set of solutions. It can be cleaned, which corresponds to reduce it to a solved form, and reused for further computations: this method is incremental.

We did not precise so far on which objects the automata are working, which depends on the representation of the constraints domain values by words, trees or graphs. The idea is the same in each case: the class of automata has to possess several closure properties and emptiness should be decidable. Several automata

Features are constructed from three given sets, a $\mathcal{S}$ set of sorts (also called basic types), a set $\mathcal{K}$ of keys (also called features), and a set $\mathcal{X}$ of sort variables. The set $\mathcal{F}$ of feature expressions (or terms) follows the syntax given in the form of the following context free grammar:

$$
\begin{array}{lcccc}
\mathcal{F} & := & \mathcal{T} & \mid & \mathcal{T}(\mathcal{L}) \\
\mathcal{T} & := & \mathcal{S} & \mid & \mathcal{X} \\
\mathcal{L} & := & \mathcal{K} \Rightarrow \mathcal{F} : \mathcal{T} & \mid & \mathcal{K} \Rightarrow \mathcal{F} : \mathcal{T}; \mathcal{L}
\end{array}
$$

For example the feature term $person(age \Rightarrow X : nat)$ is meant to represent the set of all persons whose age is the natural number X. For our purpose, feature constraints will be conjunctions of equalities between feature terms. For example, one can specify as a constraint the set of all persons who have blue eyes, and are married with a person of the same age:

$$
\begin{array}{l}
X = person(age \Rightarrow Z : nat; eyes \Rightarrow blue : colour; married \Rightarrow Y : person) \\
\& \\
Y = person(age \Rightarrow Z : nat)
\end{array}
$$

Unlike ordinary trees, feature terms may have an arbitrary number of subterms for each sort name. Still, feature terms unification looks pretty much like ordinary unification. Let us give the decomposition and conflicting rules only:

Decompose

$$x(Key \Rightarrow y : z) \; = \; x'(Key \Rightarrow y' : z') \quad \rightarrow \quad x = x' \;\&\; y = y' \;\&\; z = z'$$

KeyConflict

$$x(Key_1 \Rightarrow y : z) \; = \; x'(Key_2 \Rightarrow y' : z') \quad \rightarrow \quad \perp$$

if Key_1 and Key_2 are different

SortConflict

$$s = s' \quad \rightarrow \quad \perp$$

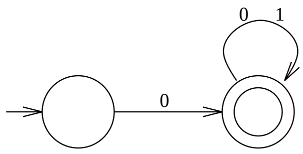
if s and s' are different sort names

In these rules, x, y, z, x', y', z' stand for constants or variables of the appropriate categories. For example, x may be the sort *person* and y the variable X above. As an example of use, the constraint:

$$
\begin{array}{l}
X(age \Rightarrow I : nat) = person(age \Rightarrow J : nat) \\
\rightarrow_{Decompose} \\
X = person \;\&\; I = J
\end{array}
$$

confines X to describe a set X of persons having the same age.

Figure 4. Feature constraints



Once we read a 0, then we know that the number (which is written in base 2) is even. Hence we enter a final state marked with a double circle. Then, whatever we read, the number will be even and we stay in that final state.

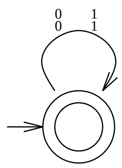
Figure 5. An automaton accepting even numbers

have been designed in the literature, each solving some particular kind of constraint. Let us mention the classical finite state automata on words, the finite tree automata, the automata with equality and disequality constraints, the tree automata with free variables, the tree matching automata,... All possess the appropriate closure properties by set operations (see e.g. [8] for more details).

The efficiency of automata techniques will follow directly from the efficiency of operations on automata and emptiness decision. To get an idea of the computational complexity, let us recall that, for classical word automata, union is computed in constant time, intersection in quadratic time, projection in linear time and complement in linear time for deterministic automata and exponential time for non-deterministic ones. Emptiness decision is linear. This behaviour can be found again in most automata classes: determinization and complement are usually the most expensive steps.

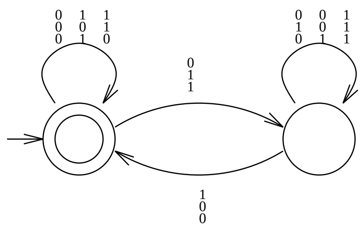
Let us give a few examples of constraint solving using automata. Presburger arithmetic consists in first-order formulae over the atomic formulae $s = t$ and $s \geq t$ where s, t are built using constants $(0, 1, 2, \ldots)$ addition and multiplication by a constant. The interpretation domain is the set of natural numbers. For instance, $\exists x \; y = 2 * x$ defines the set of even numbers. Assume now that the natural numbers are written in base two, from right to left. For example, the number thirteen will be written 1011. With each formula can be associated a word automaton. For example, $\exists x \; y = 2 * x$ is associated with the automaton of figure 5. In general, the construction might be a bit more complicated because some formulae may have $n > 1$ free variables. In such a case, the n-uples of numbers are encoded as words over $\{0, 1\}^n$. For example, the pair (thirteen, four) will be represented by the word $\begin{smallmatrix}0&0&1&0\\1&0&1&1\end{smallmatrix}$: reading from right to left the lower word we find thirteen in base 2 and reading from right to left the upper word we find four. The height of the stack of numbers is the number of free variables.

With such a convention, it is possible to build the automaton accepting the pairs of numbers which satisfy $x = y$ (see figure 6) and the set of triples x, y, z of numbers



The only possible transitions are those which are labeled with pairs of identical symbols.

Figure 6. An automaton for the formula $x = y$



The two states correspond to "no carry" (the final state) and "carry" the non-final state. Initially we enter without carry. Since 0+1 = 1, 0+0 = 0, 1+0=1, reading the triples $\begin{smallmatrix}0\\1\\1\end{smallmatrix}$, $\begin{smallmatrix}0\\0\\0\end{smallmatrix}$, $\begin{smallmatrix}1\\0\\1\end{smallmatrix}$, we stay in the final state. Since $1 + 1 = 0$ with carry 1, we have the transition by $\begin{smallmatrix}1\\1\\0\end{smallmatrix}$ to the state carry. All other transitions are built in the same way.

Figure 7. An automaton for the formula $x = y + z$

such that $x = y + z$ (see figure 7). Now, combining these two automata, by intersection and projection (this last operation simply forgets a component of the tuples), we get back the automaton of figure 5.

Automata techniques have been used for a large variety of constraints, but few of them are really used in actual software, either because of their high computational complexity or because of their relatively recent discovery. A simple example is the use of tree automata in membership constraints, which express typing properties; the formulae consist in combinations of atomic formulae $t \in \zeta$ where ζ is recognized by a finite tree automaton. For example, we might want to restrict to integers the values that a given variable x can take. Assuming that we have the function symbols $0 :\rightarrow \mathsf{int}$, $+ : \mathsf{int} \times \mathsf{int} \rightarrow \mathsf{int}$, $E : \mathsf{real} \rightarrow \mathsf{int}$, ... We can express the typing condition using a constraint $x \in \mathsf{int}$ where int is interpreted as the set of trees recognized in the state int by the above specified automaton. This will include $0 + E(t)$ where t is a real value, but not $t + E(t)$. Such constraints are used, e.g. in the constraint concurrent logic programming language Oz [47], developed at DFKI in Saarbrücken. Typing constraints can also be inferred at compile time in a logic programming language, taking advantage of the structure of the program [19]. Then automata techniques can be devoted to this particular piece of program, which yields more efficient execution. The first-order theory of typing (also called membership) constraints has been proved decidable by H. Comon and C. Delor [10].

Related to typing constraints, *set constraints* have been introduced by Mishra in 84 [40] in the area of program analysis. Set constraints are combinations of formulae $e \subseteq e'$ where e, e' are *set expressions* using e.g. intersection, union, complement, application of a function symbol, etc... They are interpreted as (possibly infinite) subsets of the set of all terms. The contribution of the French school in this area was to demonstrate that some kind of tree automata (*tree automata with free variables*) are adequate for the representation of solutions and for constraint solving [15], [22, 21, 51]. Set constraints can also be used as part of a constraint logic programming language and there are other techniques than tree automata to solve them [34, 5]

3.2. Other algebraic representations

Besides automata techniques there are other (complete) constraints methods which extensively use the representation of the constraints. A first example is *equational unification* as already introduced in section 2, and we will consider here the case where there are associative and commutative function symbols. An alternative to the syntactic methods already described is to associate a semi-ring [2] with the equational axioms, and then to reduce unification constraints to the solving of equations in the semi-ring. Let us also mention an old semantic algorithm by Löwenheim for Boolean ring unification which has been rediscovered by Martin and Nipkow [38]. Another example of importance concerns the linear diophantine equations that consist of linear polynoms with integer coefficients. They are used in particular in another constraint solving problem, which again uses semantic methods: solving equations between terms in presence of associative and commutative function symbols.

A typical example of diophantine equations is the system:

$$\begin{cases} -x_1 + x_2 + 2x_3 - 3x_4 = 0 \\ -x_1 + 3x_2 - 2x_3 - x_4 = 0 \end{cases}$$

where we are searching only the positive integer solutions. We have seen already that this can be solved by means of automata. However, in this particular situation, specific methods can be more efficient. The idea of Contejean and Devie [13] is to represent the solutions as vectors in the real space of dimension n (the number of

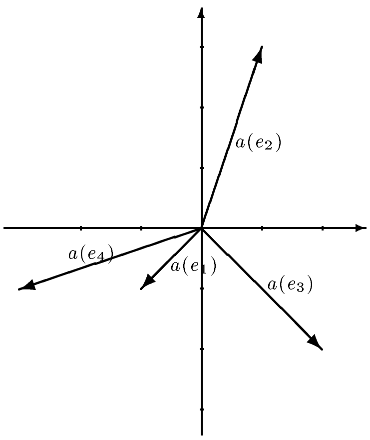


Figure 8. Default vectors

equations, 2 in our example). The constraint solving algorithm then consists of starting from the origin and repeat the addition of one of the *default vectors* until we reach back the origin. Default vectors are computed from the equation system as the values of the two left members of the equations on the vector basis. In our example, the default vectors are represented on figure 8. After the first step, the addition of default vectors d to the current vector v is only considered when the scalar product $d \cdot v$ is negative. Then the process is shown to be always terminating and complete, in the sense that every solution of the system will be a combination of the solutions obtained in this way. For example, figure 9 shows how the solution (4,2,1,0) is reached by the algorithm. 4 is the number of times we have added the default vector $a(e_1)$, 2 is the number of times we have added the default vector $a(e_2)$, 1 is the number of times we have added the default vector $a(e_3)$, 0 is the number of times we have added the default vector $a(e_4)$.

Other semantic methods have been proposed for linear Diophantine equations solving; see e.g. [1, 18, 17, 43, 50].

4. Hybrid Methods

As we have seen in the previous sections, solving a constraint consist in finding some "canonical representation" of it. It is canonical in the sense that all unsatisfiable constraints have a unique representation. This corresponds to what we call *complete constraint solvers.* The completeness is not really required in many situations. Consider for instance applications of constraints in automated deduction or logic programming. Constraints allow to represent (in an efficient way) sets of

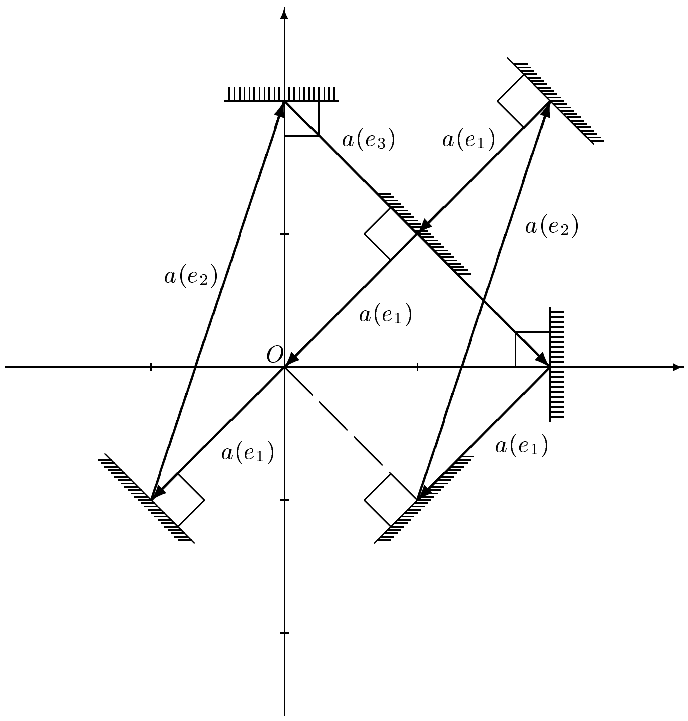


Figure 9. Computation of the solution (4,2,1,0) of the system

formulas. Then inferences take constrained formulas as premisses and yield constrained formulas as conclusion. As long as the proof is not completed, or as soon as a contradiction is not found (the empty clause in logic programming), it does not matter for the deduction process whether or not the constraint of a formula is satisfiable or not. Of course, not checking for satisfiability may yield a huge number of junk formulas. But, on the other hand, checking satisfiability can be quite expensive in time. That is why in most constraint logic programming languages or constraint automated deduction systems do not solve the constraints eagerly; there is a trade-off between time efficiency and space efficiency. Usually some constraint simplifications are performed eagerly and the complete constraint solving is performed only when the empty clause is generated (or, more generally when a proof is obtained).

As a typical example, the incomplete solver checks, at each step of the computation, a *relaxed* version of the set of constraints, an idea quite common in Operations

Research, where the name of *relaxation* comes from. In practice, rather than checking a conjunction of a large number n of elementary constraints, a possibility is to check all possible combinations of conjunctions of two elementary constraints, a technique known as *local consistency.* Since local consistency does not imply the existence of solutions, this technique is usually used in combination with an *enumeration procedure* operating on locally consistent constraints in order to restore the completeness of the whole constraint solving process. Enumeration should be taken here in a broad sense: relations as well as values may be enumerated.

A *hybrid method* is therefore a constraint solving technique which comes in two parts: one syntactic constraint solving technique, which is incomplete in general, and a (syntactic or semantic) technique which completes the first part of the constraint solving. While the first simplification is performed eagerly, the second part of the constraint solving is performed only when necessary.

Hybrid methods are actually used in all applications of constraints in automated deduction and logic programming, as soon as the constraint solving is expensive. In logic programming, a typical example is the finite domain constraints. In automated deduction, this is the case of *ordering constraints* [20]. *Combination mechanisms* can also be seen as an instance of this scheme. Let us consider now these constraints in more detail.

4.1. Finite domains

Finite domain constraints are described in figure 10, together with a constraint solving method based on local consistency checking. These techniques are at the root of the success of Constraint Programming environments like CHIP [**?**] or Ilog Solver [**?**] for solving combinatorial problems. In practice, the efficiency of these hybrid methods for finite domains heavily depends on the enumeration procedure for which various heuristics can be used. Although some of them work pretty well, such as the *first-fail* principle for variables and *domain-splitting* for values in the context of finite domains constraints, there is no best strategy applicable for all types of problems.

Finite domain constraints as they are defined in figure 10 do not allow to easily specify and solve complex problems in areas like planning, scheduling, packing and placement. To this end, new predicates (like $\neq$) and logical connectives (like negation and disjunction) can be introduced to the price of efficiency problems. Moreover, these particular predicates are too low-level to allow modelling problems in an easy and natural way. An alternative, first explored in CHIP with the *cumulative* constraint, is to introduce the so called *global constraints* to reach more expressivity by means of high-level abstractions. The cumulative constraint generalises the disjunctive constraints in order to model finite capacity scheduling problems for which there may be several copies of each given kind of resource or these resources can be shared. The second goal of these global constraints is to take advantage of the interactions among different constraints at runtime in order to reach a better pruning of the search tree. Instead of just using values or bound propagation as in the previous method, global constraints take into account structural properties of

We consider here a structure FD whose domain is the set $\mathbb{N}$ of natural numbers, hence variables will range over $\mathbb{N}$. In this setting, *Finite domain* constraints are existential positive formulae built up with the five predicates $=, >, \geq, <, \leq$ interpreted in $\mathbb{N}$, and infinitely many membership predicates $\in [a\ b]$, one for each finite interval $[a\ b]$ of $\mathbb{N}$. The precise syntax is given by the following grammar:

$C := T = T \mid T > T \mid T < T \mid T \geq T \mid T \leq T \mid x \in [a\ b] \mid C \ \&\ C$
$T := a \mid x \mid ax \mid T + T \mid T - T$
where x denotes a variable ranging over $\mathbb{N}$, a and b natural numbers. The atomic constraint $x \in [a\ b]$ is interpreted by the empty set when $b < a$. The atomic constraints $x = a$ and $x \in [a\ a]$ are identified. Solving finite domain constraints is both NP-complete and very important for practice, which has therefore favored the use of a practically efficient technique, *constraint propagation*, described here in the form of a set of transformation rules: each atomic ordering or equality constraint C between the variables in $\mathcal{V}ar(C) = \{x_1, \ldots, x_n\}$, whose domains are defined by membership constraints $x_i \in [a_i\ b_i]$, induces new restrictions on these domains, resulting in new membership constraints. We call *reduced domain* $RD(x_j, C)$ of the variable x_j for C, the smallest interval $[a'_j\ b'_j]$ containing all m in $\mathbb{N}$ such that the constraint obtained by substituting x_j by m in $C\ \&\ (\&_{i=1}^{i=n} x_i \in [a_i\ b_i])$ is satisfiable in FD.

Forward Checking	$x \in [a\ b]\ \&\ C\ \&\ \Gamma$	$\rightarrow$	$x \in RD(x, C)\ \&\ \Gamma$
		if	$\mathcal{V}ar(C) = \{x\}$
Look Ahead	$x \in [a\ b]\ \&\ C\ \&\ \Gamma$	$\rightarrow$	$x \in RD(x, C)\ \&\ C\ \&\ \Gamma$
		if	$x \in \mathcal{V}ar(C)$, and $RD(x, C) \neq [a\ b]$
Eliminate	$x = a\ \&\ \Gamma$	$\rightarrow$	$x = a\ \&\ \Gamma\{x \mapsto a\}$
		if	$x \in \mathcal{V}ar(\Gamma)$
Falsity	$x \in \emptyset\ \&\ \Gamma$	$\rightarrow$	$\perp$
Enumerate	$x \in [a\ b]\ \&\ \Gamma$	$\rightarrow$	$\Gamma\{x \mapsto a\}$ **or** $x \in [a+1\ b]\ \&\ \Gamma$
		if	no other rules applies

Using the *Forward Checking* rule is easy: since x is its only variable, C can be solved which modifies the original domain of x and allows to eliminate C. There are several uses of the *Look Ahead* rule whose principle is to lift the constraint from the variables to their domain. For example, the constraint $cx_1 < dx_2 + e$ entails the relation $ca_1 < db_2 + e$. We can therefore choose for b'_2 the largest natural number m such that $cm < db_2 + e$ (unless it is bigger than b_2). Consider now the constraint:

$$x \in [2\ 3]\ \&\ y \in [1\ 4]\ \&\ z \in [2\ 5]\ \&\ 4x > y + z\ \&\ 3x < y + z$$
$$\rightarrow_{Look\ Ahead}$$
$$x \in [2\ 3]\ \&\ y \in [2\ 4]\ \&\ z \in [2\ 5]\ \&\ 4x > y + z\ \&\ 3x < y + z$$

Continued on Figure 11

Figure 10. Finite domains: 1/2

Continued from Figure 10
Now, we enumerate, and we obtain first (the **or** must be understood as a non-deterministic branching):

$$y \in [2\ 4]\ \&\ z \in [2\ 5]\ \&\ 8 > y + z\ \&\ 6 < y + z$$
$$\rightarrow_{LookAhead}$$
$$y \in [2\ 4]\ \&\ z \in [3\ 5]\ \&\ 8 > y + z\ \&\ 6 < y + z$$

which again needs applying *Enumerate* until we get the solution:

$$x = 2\ \&\ y = 2\ \&\ z = 5.$$

We can see the crucial role of enumerations, and it is the case here that enumerating from the largest values would speed up the process. A popular schema, *Domain splitting*, is similar to binary search.

Figure 11. Finite domains: 2/2

the constraints in order to deduce and propagate more information. This is usually done via a semantic representation of the constraints based on graphs. While simple arithmetic and logic knowledge is used in the former methods, deeper finite mathematics, graph theory and operations research knowledge is required and used in the latter. Other global constraints, like "diffn", a generalization of disequality constraint to n-dimensional objects, have been recently introduced in CHIP [**?**] which permitted to solve quite difficult placement problems.

4.2. Ordering constraints

Symbolic constraints also may use enumerations in the form of a domain splitting rule since the Herbrand domain is usually infinite. This is the case with ordering constraints, conjunctions of atomic constraints of the form $s > t$ or $s = t$. They have been studied by Comon [9], Jouannaud and Okada [31], and Nieuwenhuis [41]. In this context, simplification rules are obtained by expressing the recursive definition of the ordering on terms as a set of transformation rules operating on atomic constraints, while enumeration rules linearize the constraint considering all possible orderings between the componebts. SATURATE, a system developed by Nieuwenhuis and Ganzinger [20], implements constraints solving mechanism of this form, with a lazy enumeration rule.

4.3. Combination techniques

Another kind of hybrid method arises when an application uses complex constraints operating on several computational domains, each of them being supposed to have its own syntax and constraint solving method. Such problems are called *combina-*

tion problems. In this case, the constraint syntax is of course built with different pieces coming from the different constraints involved in the combination. And, of course, it is desirable to modularly derive a constraint solving algorithm for the combination from the algorithms already known for the elementary constraints. It turns out that this problem was thoroughly investigated for the unification case, since it was actually the main difficulty of associative commutative unification: when there are associative commutative function symbols, the problem can be seen as a combination of several unification algorithms, one for each of these symbols.

Querying a bibliographic data base for articles satisfying certain constraints is an interesting practical example of application for this kind of techniques. Assume each item in the data base records information about a particular article, the authors, the title, the affiliation, etc..., as in the bibtex format. Each item can be represented by a so called feature term, and the whole (finite) data base becomes a (big) conjunction Φ of feature terms. Note that it will be easy to augment the data base, by adding a new element in the conjunction. Querying the data base for a particular entry, for example all papers containing the word "unification" in their title, will be expressed as an entailment problem of the form Φ `entails` ϕ, if ϕ represents the query. The problem here is that the feature terms are not homogeneous, since they contain subexpressions which are strings over a certain vocabulary, which involves solving associative *pattern matching* constraints, see figure 12.

Solving a constraint over an heterogeneous domain starts with a *variable abstraction* consisting in purifying the constraints: the goal is to get a conjunction of constraint systems each of which is homogeneous. Then each constraint solving algorithm can be applied to the pure parts. Still, interactions between the homogenous parts have to be considered. This can be done through an enumeration procedure which, roughly, guesses the shared variables dependencies.

5. Application Contexts

Because constraints are a natural way to specify mathematical problems, there are a number of potential applications of the ideas presented in this paper. Numerical constraints are used in Operations Research, in Robotics, and more generally in the area of applied mathematics, physics and mechanical engineering. Symbolic constraints arise naturally in computer science applications like type verification and inference, abstract interpretations, logic programming, deduction, and artificial intelligence. We concentrate briefly now on two main applications of the tools presented in this paper: constraints for deduction and constraints for computations.

The use of constraints in deduction has recently developed into a new promising field of research, of which constraint logic programming is a specific instance. Constraints are of particular importance in theorem proving for two main reasons. Constraints allow to (possibly exponentially) reduce the number of formulae resulting from a single deduction step, by packing them all in a constrained formula. For example, automated deduction techniques require solving equations between terms possibly involving associative and commutative operators, as subgoals of a given logical deduction. But, an equation as simple as $x + x + x + x = u + v + w + z$

We will use the syntax of features as defined in figure 4. Our sorts (or types) will be the elements set $\{article, person, identity, university, address\}$, augmented by "built-in types", *list-of[]*, *string* and *Nat*. Using a set of keys that are clear from the context, here is the syntax of our basic types:

$$\begin{array}{rrcl rcl}
article(& key & \Rightarrow & K : string; & title & \Rightarrow & T : string; \\
 & authors & \Rightarrow & AA : \textit{list-of[person]}) & & & \\
university(& univ & \Rightarrow & U : string; & city & \Rightarrow & C : string; \\
 & adr & \Rightarrow & A : address) & & & \\
address(& city & \Rightarrow & C : string; & street & \Rightarrow & S : string; \\
 & postalcode & \Rightarrow & P : Nat) & & & \\
person(& name & \Rightarrow & N : identity; & affiliation & \Rightarrow & U : university) \\
identity(& first & \Rightarrow & F : string; & last & \Rightarrow & L : string)
\end{array}$$

When implicit, we do not recall the constant sorts in a feature term. We now consider the query:

$$\begin{array}{rrcl}
A = article(& key & \Rightarrow & X; \\
 & title & \Rightarrow & Y.\mathsf{Unification}.Z; \\
 & authors & \Rightarrow & U)
\end{array}$$

where the letters A, X, Y, Z, U denote variables of the appropriate types. The solutions of this query are all the values for A, X, Y, Z, U for which the constraint above is entailed by the data base expression. Of course, this entailment problem will cause an enumeration of all elementary formulas in the data base corresponding to the various bibliographical data, allowing then to resolve the entailment problem for each bibliographical datum in turn. Assume now that the bibliographical data base is the one at the end of this paper and thus contains the following entry:

$$\begin{array}{rrcl}
article(& key & \Rightarrow & \mathsf{MartelliMontanari82}, \\
 & title & \Rightarrow & \mathsf{An\ Efficient\ Unification\ Algorithm}, \\
 & authors & \Rightarrow & [person(name \Rightarrow ident(first \Rightarrow \mathsf{Alberto}, \quad) \\
 & & & \qquad\qquad\qquad\qquad\qquad last \Rightarrow \mathsf{Martelli})) \\
 & & & \quad person(\ name \Rightarrow ident(first \Rightarrow \mathsf{Ugo}, \\
 & & & \qquad\qquad\qquad\qquad\qquad last \Rightarrow \mathsf{Montanari}))]
\end{array}$$

In order to make the variables match the appropriate information, the constraint will first be split into homogeneous pieces, that is pieces of a given domain. Here, there are three domains, feature terms, strings, and natural numbers. We therefore obtain the new constraint:

$$\begin{array}{rcl c rcl}
key & \Rightarrow & X & = & key & \Rightarrow & \mathsf{MartelliMontanari82} \\
title & \Rightarrow & Y.\mathsf{Unification}.Z & = & title & \Rightarrow & \mathsf{An\ Efficient\ Unification\ Algorithm} \\
authors & \Rightarrow & U & = & authors & \Rightarrow & \\
 & & & & \multicolumn{3}{l}{[person(name \Rightarrow ident(first \Rightarrow \mathsf{Alberto},}} \\
 & & & & \multicolumn{3}{l}{\qquad\qquad\qquad\qquad last \Rightarrow \mathsf{Martelli}))} \\
 & & & & \multicolumn{3}{l}{\ person(name \Rightarrow ident(first \Rightarrow \mathsf{Ugo},}} \\
 & & & & \multicolumn{3}{l}{\qquad\qquad\qquad\qquad last \Rightarrow \mathsf{Montanari}))]}
\end{array}$$

which in turn is decomposed in:

$X = \mathsf{MartelliMontanari82}\ \&\ Y = \mathsf{An\ Efficient}\ \&\ Z = \mathsf{Algorithm}\ \&\ U = ...$

by using an appropriate string matching algorithm.

Figure 12. Bibliographic search

has over 30 billions of incomparable solutions. It is of course not realistic to engage current computers in such a number of further deductions. The remedy, keeping the equation as a constraint for the further deductions, was initiated in [33]. Constraints allow also to record information on the computation as a component of the deduced formulae, hence they permit to formulate inference rules which reason about this information for efficiently pruning the search space.

Constraint solving techniques have found their first key application within the Constraint Logic Programming framework, especially for solving Combinatorial Search Problems [26]. As examples of these problems occurring in different economical areas, we can mention project management, production scheduling, crew assignment, tour planning, etc. Not only are these problems NP-hard in general, they are also hard to model and therefore hard to program. Constraint Logic Programming brings an ideal solution to this problem by supporting different types of constraint systems (symbolic and numeric), and by allowing the use of powerful constraint solving algorithms combined with domain-oriented heuristic search. Several successful industrial applications have been developed in these areas, which are described in this volume. An overview of Constraint Logic Programming can be found in [27], where are also described applications to circuit design (symbolic verification, test pattern generation, logic synthesis, diagnosis) and decision problems in management (option trading analysis, portfolio management).

Acknowledgments

This work was partly supported by Vincent Mongeard-Mugneret, 21700 Vosne Romanée, France, Fax: 33-80-62-35-75 and by the ESPRIT BRA CCL. We thank Carlos Castro for his comments on earlier version of this paper.

Notes

1. Vincent Mongeard-Mugneret actually prefers supreme quality to big profit, hence uses manure only. We recommand his *Clos Vougeot Grand Cru.* 88, 89 and 90 are extremely good years. Feel free to use the 3rd author's recommandation.
2. A semi-ring is a structure which has the same properties as a ring, except that there might be no inverse for addition. $(\mathbb{N}, +, \times, 0, 1)$ is a typical example of a semi-ring.

References

1. H. Abdulrab and M. Maksimenko. General solutions of systems of linear diophantine equations and inequations. In Jieh Hsiang, editor, *6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, Kaiserslautern, Germany, April 1995. Springer-Verlag.
2. H. Aït-Kaci and A. Podelski. Towards a meaning of life. In J. Maluszyǹski and M. Wirsing, editors, *3rd International Symposium on Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 255–274, 1991.
3. R. Backofen and G. Smolka. A complete and recursive feature theory. *Theoretical Computer Science*, 146(1–2):143–268, 1995.

4. Ricardo Caferra and Nicolas Zabel. A method for simultaneous search for refutations and models by equational constraint solving. *Journal of Symbolic Computation*, 13(6):613–642, June 1992.
5. A. Cheng and D. Kozen. Some notes on rational spaces. Technical report, Comp. Science Department, Cornell University, 1996.
6. A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of FGCS'84*, pages 85–99, November 1984.
7. H. Comon. Disunification: a survey. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 9, pages 322–359. The MIT press, Cambridge (MA, USA), 1991.
8. H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. A preliminary version of this unpublished book is available on `http://l3ux02.univ-lille3.fr/tata` .
9. Hubert Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.
10. Hubert Comon and Catherine Delor. Equational formulae with membership constraints. *Information and Computation*, 112(2):167–216, August 1994.
11. Hubert Comon and Pierre Lescanne. Equational Problems and Disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.
12. Hubert Comon and Ralf Treinen. Ordering constraints on trees. In Sophie Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 1–14, Edinburgh, Scotland, 1994. Springer-Verlag. (Invited Lecture).
13. Evelyne Contejean and Hervé Devie. An efficient algorithm for solving systems of diophantine equations. *Information and Computation*, 113(1):143–172, August 1994.
14. G. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.
15. M. Dauchet. Rewriting and tree automata. In Hubert Comon and Jean-Pierre Jouannaud, editors, *Proc. Spring School on Theoretical Computer Science: Rewriting*, volume 909 of *Lecture Notes in Computer Science*, Odeillo, France, 1994. Springer-Verlag.
16. M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 693–702. Institute for New Generation Computer Technology, 1988.
17. Eric Domenjoud. Outils pour la déduction automatique dans les théories associatives-commutatives. Thèse de doctorat de l'université de Nancy I, 1991.
18. Miguel Filgueiras and Ana Paula Tomás. A fast method for finding the basis of nonnegative solutions to a linear Diophantine equation. *Journal of Symbolic Computation*, 19:507–526, 1995.
19. Thom Frühwirth, Ehud Shapiro, Moshe Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proc. 6th IEEE Symp. Logic in Computer Science, Amsterdam*, pages 300–309, 1991.
20. H. Ganzinger and R. Nieuwenhuis. *The Saturate System*, 1996. This manual is available at `http://www.mpi-sb.mpg.de/ hg/systems/SATURATE/Saturate.html` .
21. R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated subset relationships. In *Proc. 34th Symposium on Foundations of Computer Science*, pages 372–380, Palo Alto, CA, November 1993. IEEE Computer society press.
22. Rémy Gilleron, Sophie Tison, and Marc Tommasi. Solving systems of set constraints using tree automata. In *Proc. 10th Symposium on Theoretical Aspects of Computer Science, Würzburg, LNCS*, 1993.
23. N. Heintze. *Set based program analysis.* PhD thesis, Carnegie Mellon University, 1992.
24. J. Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Soc. des Sciences et des Lettres de Varsovie, Classe III*, 33(128), 1930.
25. Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots \omega$. Thèse d'Etat, Univ. Paris 7, 1976.
26. J. Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th Annual ACM Symposium on Principles Of Programming Languages, Munich (Germany)*, pages 111–119, 1987.

27. Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–582, 1994.
28. J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.
29. Jean-Pierre Jouannaud, editor. *First International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, München, Germany, September 1994. Springer-Verlag.
30. Jean-Pierre Jouannaud and Emmanuel Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1), July 1989.
31. Jean-Pierre Jouannaud and Mitsuhiro Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In Javier Leach Albert, Burkhard Monien, and M. Rodriguez Artalejo, editors, *18th International Colloquium on Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*, pages 455–468, Madrid, Spain, 1991. Springer-Verlag.
32. N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.
33. Claude Kirchner, Helene Kirchner, and Michaël Rusinowitch. Deduction with symbolic constraints. *Revue Française d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on automatic deduction.
34. D. Kozen. Set constraints in logic programming. *Information and Computation*, 1995. To appear.
35. Jean-Louis Lassez and K. G. Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3(3):1–17, September 1987.
36. M. J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd IEEE Symp. on Logic in Computer Science*, pages 348–357, July 1988.
37. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
38. U. Martin and T. Nipkow. Boolean unification. the story so far. *Journal of Symbolic Computation*, 7(3 & 4), 1989.
39. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In P. Schroeder-Heister, editor, *Extensions of Logic Programming.* LNCS 475, Springer Verlag, 1991.
40. P. Mishra. Towards a theory of types in prolog. In *1st IEEE Symposium on Logic Programming*, pages 456–461, Atlantic City, 1984.
41. Roberto Nieuwenhuis. Simple LPO constraint solving methods. *Information Processing Letters*, 47(2):65–69, August 1993.
42. A. Podelski, editor. *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
43. L. Pottier. Minimal solutions of linear diophantine systems : bounds and algorithms. In *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications*, pages 162–173, Como, Italy, April 1991.
44. W. C. Rounds and R. Kasper. A complete logical calculus for record structures representing linguistic information. In *Proc. 1st IEEE Symp. Logic in Computer Science, Cambridge, Mass.*, June 1986.
45. M. Schmidt-Schauß. An algorithm for distributive unification. Research Report 13/94, Fachbereich Informatik, Univ. Frankfurt, 1994.
46. G. Smolka. The oz programming model. In J. van Leeuwen, editor, *Current Trends in Computer Science*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343, 1995.
47. Gert Smolka. A foundation for higher-order concurrent constraint programming. In Jean-Pierre Jouannaud, editor, *Proceedings of the 1st International Conference on Constraints in Computational Logics, Munich (Germany)*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72. Springer-Verlag, 1994.
48. M. Tajine. The negation elimination from syntactic equational formulas is decidable. In C. Kirchner, editor, *Proc. 5th Rewriting Techniques and Applications, Montréal, LNCS 690*, 1993.

49. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 134–191. Elsevier, 1990.
50. A. P. Tomás and M. Filgueiras. Solving linear diophantine equations using the geometric structure of the solution space. In H. Comon, editor, *Proc. 8th Conf. Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*, pages 269–283, 1997.
51. Marc Tommasi. Automates et contraintes ensemblistes. Thèse de l'Univ. de Lille, February 1994.
52. R. Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–458, November 1992.
53. Ralf Treinen. Feature trees over arbitrary structures. In Patrick Blackburn and Maarten de Rijke, editors, *Specifying Syntactic Structures*, chapter 7, pages 185–211. CSLI Publications and FoLLI, 1997.