A Methodological View of Constraint Solving

Hubert Comon[†] **Mehmet Dincbas**[‡]
Jean-Pierre Jouannaud[†] **Claude Kirchner**[§]

Abstract

Constraints have become very popular during the last decade. Constraints allow 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. Examples are used throughout the paper to illustrate the concepts and methods. We also discuss applications of constraints to various fields, such as programming, operations research, and theorem proving.

[†] CNRS and LRI, Bat. 490, Université de Paris Sud,
91405 ORSAY Cedex, France
{comon, jouannaud}@lri.lri.fr

[‡] COSYTEC, Parc Club Orsay Université, 4 Rue Jean Rostand,
91893 Orsay Cedex, France
dincbas@cosytec.fr

[§] INRIA Lorraine & CRIN, 615 rue du jardin botanique, BP 101,
54602 Villers-lès-Nancy Cedex, France
Claude.Kirchner@loria.fr

Acknowledgements: This work was partly supported by the ESPRIT BRA CCL, and by Gérard Vachet-Rousseau, 21220 Gevrey-Chambertin, FRANCE, FAX: 33-80-51-86-74.

1 Introduction

Constraints have become tremendously popular during the last decade. Constraints allow to define sets of data by means of logical formulae. For example, the set of even numbers can be described by the formula $\exists x\ y = 2 * x$, interpreted in the set of natural numbers. Such representations are familiar to users, since they correspond to the comprehensive definition of sets. The above example defines the set $\{y \in \mathbb{N} \ : \ \exists x \in \mathbb{N}\ y = 2 * x\}$. This is a first reason for the success of constraints.

The comprehension axiom allows to define sets that would not be logically definable by the extension axiom only. This is true for constraints too, of course, but there is also a more pragmatic argument in favor of constraints: they allow to express and manipulate *large* sets of data that could not be reasonably handled if given in extension.

Besides this familiar aspect that appeals to users, constraints enjoy a declarative aspect that seduces computer scientists of the academia. There are many ways of defining even numbers, such as by saying that it is the least set of natural numbers that contains 0 and is closed by taking successor twice. This definition is procedural, in the sense that it is indeed an algorithm for enumerating all even numbers starting from 0. Let's assume that odd numbers are defined similarly, as the least set of natural numbers containing 1 and closed by taking successor twice. How to express the (empty) intersection of these two sets in this setting? Using the procedural definition of the two sets, would require to enumerate them both and check whether some value is enumerated twice. This may of course not terminate. Constraints can handle this effectively: this is indeed part of the concept, as explained in the sequel.

The declarative aspect is often opposed to efficiency. A second, important reason why constraints are so popular, is that they reconcile declarativeness with efficiency. Since the domain in which a constraint computes is known in advance, very efficient algorithms can be chosen to solve it, instead of general purpose procedures as it is the case, for example, in usual logic programming languages. For example, a standard PROLOG implementation would solve a system of linear equations over the rationals by using resolution, while a language allowing for rational constraints will use Gauss elimination instead, and therefore be much more efficient.

There is yet a third reason for the success of constraints: they can be very easily combined with the logic programming paradigm, and more generally with any logic-based computation model. Besides, their use is compatible with a dynamic generation, as it is the case in most applications. This characteristic is called *incrementality*. It is of course quite difficult to identify precisely when the notion of constraint was first used explicitly in computer science. However, as far as their combination with the logic programming paradigm is concerned, they play a key role since their introduction in PROLOG II [3], their general formulation by Jaffar and Lassez [13], and the introduction of finite domain constraints in CHIP [9].

So far, we have not explained a major aspect of constraints, which is directly related with the title of this paper. There are (in general infinitely) many ways to represent a given set of data by means of constraints. For example, the set of even numbers could also have been defined as the set of natural numbers which are not odd, that is by the constraint $\forall x \in \mathbb{N}\ (\exists y \in \mathbb{N}\ x = 2 * y + 1) \Rightarrow z \neq x$. 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, in which case it is surely more complicated than the trivial expression for the empty set, the formula $\bot$, the usual abbreviation for *false*. Indeed, it is a major problem to avoid useless computations operating on constraints defining empty sets. So, an algorithm for deciding whether a constraint defines the empty set is a major requirement.

To summarize this informal discussion, 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 conjunction (denoted $\&$). 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*.

Consider as an example conjunctions of linear equations over rational numbers, such as $y = \frac{4}{3}x \ \& \ x + y = 63$.

2. The structure $\mathcal{M}$ in which the formulae are interpreted. This structure is made of a domain of values on one hand, and operations and relations on the other hand, interpreting all symbols in the syntax.

   In the above 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 of $\mathcal{M}$ 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 computes a particular representation of the set of all solutions of any constraint C in the structure $\mathcal{M}$. 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'.

   In the above example, many procedures are known to solve linear equations over rational numbers.

Operations research is an important area of application. We will later describe in more details constraints on finite domains. The problem here is that 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 many other applications in computer science, where the most important domain is itself made of expressions, called *terms* in this context. A typical example is the most well-known problem of unification of terms, as it appears, e.g. in classical logic programming. Unification is nothing but solving conjunctions of equations over the set of terms, as we explain in section 2. We call *symbolic*, constraints over terms [4, 16].

Symbolic constraints have been used for years in computer science, with the following applications:

- 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 a regular (tree) grammar or equivalently recognized by a bottom-up tree automaton.

- to 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.

- to represent infinite terms, the structure used by most implementations of logic programming languages, at least those which do not use the so-called occur check. For example, the infinite list of zeros can be represented by the constraint $x \in List(\mathbb{N}) \ \& \ x = cons(0, x)$.

- to express strategies in automated deduction at the object level in contrast to the meta-level, allowing to prune further the search space.

The need for more complex *combined constraints* involving several domains of interpretation arises in many applications. 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 details 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 some language combining feature constraints for expressing record definitions, with word constraints for expressing the search of a given word.

Listing all constraint systems which have ever been used would be an impossible task. Instead, we will organize this paper around the notion of constraint solving method. We distinguish among: (complete) *syntactic methods*, which do not commit to any particular representation of the problem domain nor 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 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

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.

Figure 1 illustrates these concepts by developping an example of solving constraints over rationals. This technique is attributed to Gauss, the celebrated German mathematician of the 19th century, but was actually in use before. An apparently 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, where 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, together with the general principles of the *simplex* algorithm invented by Dantzig [7]. Although this is an exponential algorithm, it is actually prefered to the recent polynomial algorithms such as the "interior point methods" [17] for two main reasons. It performs quasi linearly in the average, and has an efficient incremental version.

Remark that using any strategy of eliminating variables from a set of linear equations will eventually yield the result. The situation is quite different with the simplex method, for which there are looping sequences, but hopefully, there are also terminating stategies. Although these strategies are non-deterministic, they can be made deterministic by using a total *a priori* given order on variables, which can be used for guiding the choice of pivoting variables at each step.

These examples show all the ingredients of the transformation method. The constraint is a logical formula. Each rule transforms a constraint into an equivalent one. The equivalence proof is trivial

[1] Gérard Vachet actually prefers supreme quality to big profit, hence uses manure only. We recommand his *Mazis Chambertin Grand Cru.* 88, 89 and 90 are extremely good years. Don't mind using our recommandation.

here, but may sometimes be difficult since it amounts to show that each transformation relies on some algebraic property of the constraint domain. The solved form is obtained when no transformation applies anymore, that is, it is a normal form with respect to the transformation rules. This assumes a termination proof, at least for some particular strategy or strategies. 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 $\bot$.

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) *Completeness*: there are enough rules, that is, for each constraint which is not a solved form, there is at least one rule which applies. As a consequence, the algorithm never fails to detect that a given constraint has no solution. This property is usually simple to prove, since new rules may be added when it is not satisfied.

   (c) *Uniform* (resp. *weak*) *termination*: given an arbitrary constraint, every (resp. some fixed) sequence of transformation originating from it results in a solved form after a finite number of steps. This property may be hard to prove. For the case of weak termination, the sequence of transformations should of course not depend upon the input. Characterizing such a sequence when uniform termination is not satisfied may be a difficult task.

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.

This approach was indeed sucessfully used for a variety of problems, starting with J. Herbrand [11] 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 [20], who formulated it from a more operational point of view. The presentation of figure 3 is borrowed from [16].

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 the **Decompose** rule is no longer correct. For example, if the head symbol f in the rule is binary and commutative, then the right hand side should split in 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 either in general. Many investigations were boosted by the practical need of building some of these theories in the resolution based theorem provers. This is the case, for example, of commutativity and associativity which occurs in many useful algebraic structures: it was the first case systematically studied from this point of view, by Kirchner first [18], and later by Boudet. Other important axioms were also considered by many authors: distributivity by Arnborg and Tiden first, more recently by Contejean, and finally shown decidable by Schmidt-Schauss. But automated deduction was not the only application area where these techniques were eventually used: an important theory arising in the context of library

search, related to the axiomatization of cartesian closed categories, was investigated by Rittri; a theory originating from type inference for records was investigated by Remy. Remy's work was based on another work of Kirchner about syntactic theories [18], for which the decomposition schemas can be automatically obtained from an adequate presentation of the theory, allowing to give a general solution for this class. Motivated by harware verification problems, the theory of Boolean rings has been considered by Buttner and Simonis.

Roughly speaking, unification constraints are quantifier and negation free. 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* [5]. Caferra's method for building counter examples in theorem proving uses equational constraints with negations. The method by Jouannaud and Kounalis for inductive theorem proving needs equational constraints with universal quantifiers and negations. This is also the case of some inductive inference problems studied by Lassez and Marriot. It is actually the case that any equational constraint can be reduced to a solved form, as shown independently by Maher and Comon. 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 was given by Tajine to this important problem which has applications to compilation of rewrite rules, 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 is important. Unfortunately, it is undecidable to know whether an arbitrary equational constraint has solutions in this particular case, and this is true as well for most algebraic structures. However, one case was solved positively by Comon, Haberstrau and Jouannaud, the case of syntactic theories. However, their pratical interest is not as high as their theoretical one.

There are other important predicates in practice. Term orderings, for example, surface naturally in automated deduction where they are used to reduce the search space by eliminating those inferences which do not satisfy certain reducing condition. Membership predicates are important in typed languages where they are interpreted as set inclusion. Set predicates have been used for defining and computing partial interpretations of logic programs. All these cases were investigated thoroughly, by many authors, including Comon and Jouannaud. Again, the problem becomes more difficult, in general undecidable, when some operators satisfy certain equational axioms.

First-order terms (that we have used up to now) lack the possibility to use and bind functional variables. For doing so the λ-binder is used in higher-order logic and functional programming. Unfortunately, even unification constraints become undecidable for the so-called *lambda terms*. However, they are semi-decidable, and it turns out that Huet's semi decision procedure [12] is used in most proof development systems. An important decidable subcase of terms called pattern is used in Miller's 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 [22], and more generally in knowledge representation, under the name of *features*. They have been used also together with subtyping mechanims in the area of programming languages to model inheritance [2]. 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 mechanism for the first-order theory of features on one hand, and in many prototype implementations of feature-based languages on the other hand.

Finally let us note that most of the efforts in symbolic constraint solving are surveyed in [16, 4] to which the reader is referred for more details. See also [15] for more recent work.

3 Semantic Methods

Semantic methods, as opposed to syntactic ones, do not simply rely on a translation of the constraint syntax. They rather use another representation of the solutions of the constraint and use this specific data structure.

A typical example is the use of automata in constraint solving. This method is described e.g. in the survey of Max Dauchet [8]. The idea, which goes back to Büchi in the early sixties, consists in associating with each formula an automaton accepting the solutions of the constraint. A (finite state) automaton is a very simple computation model which includes a finite memory (the *states*) and a finite control (*transition rules*). An automaton moves from a state to another depending on the input symbol and the transition rules. An automaton cannot backtrack on its input and has no output device. It visits only once each input symbol.

Once an automaton has been associated with each atomic constraint, the logical connectives correspond to set operations on the 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. Assume that the class of automata under consideration is closed under intersection (which is the case for most known automata classes). Then 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. This is actually a general rule: each logical connective corresponds to a closure property for automata: conjunction yields intersection, disjunction yields union, negation yields complement, existential quantification yields projection. Actually, this last operation, though algorithmically very simple, has to be explained a little bit further: the automata have to recognize *tuples* of words (or trees or graphs) with as many elements in the tuple as free variables in the constraint. We will come back later to this problem.

Now, once an automaton A is associated with the constraint C, the satisfaction problem for C is equivalent to emptiness decision for A: 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 to solved forms, and reused for further constraint solving: this method is incremental.

We did not precise so far on which objects the automata are working. Actually, it depends on the representation of the constraints domain elements: words, trees, graphs. The idea is the same in each case: the class of automata has to enjoy several closure properties and emptiness should be decidable. Several automata have been designed in the literature, each solving some particular kind of constraint. Let us mention (among others) 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 ... we refer to [8] for more details.

The efficiency of automata techniques will follow directly from the efficiency of operations on automata and emptiness decision. In order 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 (resp. exponential time in the worst case) for deterministic (resp. non-deterministic) automata. Emptiness decision is linear. This behaviour can be found again in most automata classes; the most expensive step is usually the reduction of non-determinism (when it is possible) or the complement.

It is time now to give explicit examples of constraint solving using automata. Consider for instance Presburger arithmetic; the formulae are first-order formulae (say, using "and", "or", "not" and the existential quantification) over the atomic formulae $s = t$ and $s \geq t$ where s, t are built using constants (0,1,2...) 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 such that $x = y + z$ (see figure 7). Now, combining these two automata, by intersection and projection (this operation forgets a component of the tuples; it corresponds to existential quantification) 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 softwares, either because of the high computational complexity or because of their relatively recent discovery in the area of constraint solving. For example, tree automata with equality and disequality constraints have been used successfully to solve *encompassment constraints*. Such constraints are related to term rewriting. We refer to [8] for more details.

A simpler example is the use of tree automata in typing constraints; the formulae consists 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 $0 :\to \mathsf{int}$, $+ : \mathsf{int} \times \mathsf{int} \to \mathsf{int}$, $E : \mathsf{real} \to \mathsf{int}$, ... 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 [23], 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. 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 Comon.

Related to typing constraints, *set constraints* have been introduced by Mishra in 84 in the area of program analysis. These constraints correspond to an approximation of programs and allow to infer types at compile time and to detect inconsistencies in the programs. We refer to [10] for more details. 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, see [8], 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. Set constraints were also used by Kozen as part of a constraint logic programming language [19].

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 with the equational axioms, and then to reduce unification constraints to the solving of equations in the semi-ring. This method has been systematically investigated by Baader and Nutt. Let us also mention an old semantic algorithm by Löwenheim for Boolean ring unification which has been rediscovered by Martin and Nipkow. Another example of importance concerns the linear diophantine equations. 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

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

The coefficients are integers and we are looking for 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. One idea initialized for one equation by Fortenbacher and further developed in [6], is to represent the solutions as vectors in the real space of dimension n (the number of equations, 2 in our example). The constraint solving algorithm then consists in 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. In our example (borrowed from Contejean's thesis) 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 $a(e_4)$.

4 Hybrid Methods

We have seen in the previous sections complete constraint solving methods which allow to solve constraints in various computation domains such as, for example, Herbrand terms, Boolean algebras, linear rational terms. However, even if a decision procedure exists for a given computation domain, it is not always reasonable to use it at each inference step of a computation. The problem is that the complexity of *one* such step depends on the complexity of the constraint solving method, which in turn depends on the input, in contrast to conventional languages. This problem is common to all logic programming languages, but becomes crucial when constraint solving is not linear. This complexity may indeed be very high, usually exponential, and testing the satisfiability at each step of the computation becomes too expensive. This has led practically inclined people to design incomplete constraint solvers. 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 restaure 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. In practice, the efficiency of these *hybrid* methods heavily depend 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 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 Logic Programming environments like CHIP for solving combinatorial 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 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 which can hardly be solved. However, these constraints 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* in order to reach more expressivity by means of high-level abstractions. The cumulative constraint [1] 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. Besides scheduling, application areas include packing and placement. 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 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 combinatorial problems.

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 $s = t$, as studied by Comon, Jouannaud and Okada, and Nieuwenhuis and Rubio. 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 compare two given terms. SATURATE, a system developed by Nieuwenhuis and Rubio, implements constraints solving mechanism of this form, with a lazy enumeration rule.

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 *combination 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 derive a constraint solving algorithm for the combination from the algorithms already known for the more primitive 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. The technique elaborated along the years is based on the following transformation rules: A first step is aimed at splitting a combined constraint into several, homogeneous pieces. In a second, non-deterministic step, it is guessed in which homogeneous fragment a variable will take its value. Then, each homogeneous piece ϕ can be solved by using its own method, this is indeed a local consistency check, and the resulting value of a variable x is then propagated to the constraints ψ which share the variable x with ϕ, this is to ensure global consistency. The third step is repeated until all homogeneous sub-constraints are eventually solved. This technique has been adapted to a more general framework of constraints by Baader and Schulz, under a decidability condition for the positive existential fragment of each constituent constraint system. Figure 11 describes an example of its use. Similar techniques were used by Comon, Nieuwenhuis and Rubio to solve ordering constraints operating on associative commutative equivalence classes of terms. In this problem, the ordering on terms is generated from a given total ordering on function symbols, and a domain of interpretation depending on this ordering is associated to each associative commutative function symbol.

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., in a bibtex like format. Each item can be represented by a 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 $\Phi \models \phi$, 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 (but not commutative) *pattern matching* constraints. See figure 11 for details.

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 heavily 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, partial interpretations, logic programming, deduction, and artificial intelligence. In order to ease the description of constraint solving and processing, several formalisms and softwares have been designed: let us mention ELAN, developed in Nancy, a logical framework based on rewriting logic and allowing to describe transformation rules in the same way as in the examples developed in this paper, and the Constraints Handling Rules developed at ECRC as an extension of the Eclipse CLP system.

Let us now concentrate 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 only recently developed into a new promising field research, of which constraint logic programming is a specific instance. Constraints are of particular importance for theorem proving since the deduction process underlying theorem provers can greatly beneficiate of integrating constraint handling in the deduction process. The use of constraints allows in particular to prune the search space in a spectacular way. Assume again that we have an associative and commutative operator $+$, a situation which is quite likely to occur in practice, think for example to the union operator on sets. Most advanced automated deduction techniques (such as the resolution-based ones) will require solving equations between terms involving such operators, as subgoals of a given logical deduction. But, an equation as simple as $x+x+x+x = u+v+w+z$ has exactly 34 359 607 481 *minimal solutions*, as it results from a general formula given by Domenjoud. It is of course not realistic to engage current computers in such a number of further deductions. Fortunately, it turns out that this is not necessary, since what is really needed is whether there are solutions to the equations, rather than their explicit computation. So, keeping the equation as a constraint will dramatically reduce the search space on one hand, and also save time by avoiding the explicit computation of the solutions (which may need an exponential time as in the previous case). This simple observation leads to consider constrained logical formulas, hence to adapt all deduction mechanisms to this framework. A general description of deduction with constraint was first formulated by C.Kirchner, H. Kirchner and Rusinowitch. Some completeness property of this method required new proof techniques investigated by Bachmair, Ganzinger, Lynch and Snyder on one hand, and Nieuwenhuis and Rubio on the other hand. This area of research is currently very active [15].

Using these ideas, the deduction process can express structure sharing by simply recording the value of variables in the constraint part of a formula, instead of substituting it explicitly. This allowed to formally study the complexity of deductions in presence of structure sharing, yielding polynomial complexity results for various deduction processes operating on Horn clauses with equality, as shown by Lynch.

Constraint solving techniques have found their first key application within the Constraint Logic Programming framework, especially for solving Combinatorial Search Problems. 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 [14], 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). Further developments can be found in proceedings of several recent events like [24] and [21].

References

[1] A. Aggoun and N. Beldiceanu. Extending chip in order to solve complex scheduling an placement problems. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1993. Pergamon Press.

[2] H. Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures*. PhD thesis, University of Pennsylvania, 1984.

[3] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of FGCS'84*, pages 85–99, November 1984.

[4] H. Comon. Disunification: a survey. In J.-L. 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.

[5] H. Comon and P. Lescanne. Equational problems and disunification. In C. Kirchner, editor, *Unification*, pages 297–352. Academic Press, London, 1990.

[6] E. Contejean and H. Devie. An efficient algorithm for solving systems of diophantine equations. *Information and Computation*, 113(1):143–172, August 1994.

[7] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[8] M. Dauchet. Rewriting and tree automata. In H. Comon and J.-P. Jouannaud, editors, *Proc. Spring School on Theoretical Computer Science: Rewriting*, volume 909 of *Lecture Notes in Computer Science*, Odeillo, France, 1994. Springer-Verlag.

[9] 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.

[10] N. Heintze. *Set based program analysis*. PhD thesis, Carnegie Mellon University, 1992.

[11] 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.

[12] G. Huet. A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[13] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th Annual ACM Symposium on Principles Of Programming Languages, Munich (Germany)*, pages 111–119, 1987.

[14] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[15] J.-P. Jouannaud, editor. *Proceedings of the 1st International Conference on Constraints in Computational Logics, Munich (Germany)*, volume 845 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[16] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. 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.

[17] N. Karmarkar. A new polynomial–time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[18] C. Kirchner. *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles*. Thèse de Doctorat d'Etat, Université de Nancy 1, 1985.

[19] D. Kozen. Set constraints in logic programming. *Information and Computation*, 1995. To appear.

[20] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[21] A. Podelsky, editor. *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[22] W. C. Rounds and R. Kasper. A complete logical calculus for record structures representing linguistic information. In *Proceedings, Symposium on Logic in Computer Science*, pages 38–43, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[23] G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. 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.

[24] P. Van Hentenryck and S. Saraswat, editors. *Principles and Practice of Constraint Programming*. The MIT press, 1995.

Contents

List of Figures

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

$$\exists t \begin{cases} x & = & 2 * (y \perp t) \\ x \perp t & = & y \\ x & \neq & 48 \\ y & \neq & 38 \end{cases}$$

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:

$$\exists t \begin{cases} x & = & 2 * (y \perp t) \\ 2 * (y \perp t) \perp t & = & y \\ 2 * (y \perp t) & \neq & 48 \\ y & \neq & 38 \end{cases}$$

The system is then simplified according to elementary rational number theory, yielding another equivalent system:

$$\exists t \begin{cases} x & = & 2 * (y \perp t) \\ t & = & \frac{y}{3} \\ 2 * y \perp 2 * t & \neq & 48 \\ y & \neq & 38 \end{cases}$$

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

$$\begin{cases} x & = & \frac{4}{3}y \\ y & \neq & 36 \\ y & \neq & 38 \end{cases}$$

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

Figure 1: Who are they?

Gérard Vachet-Rousseau, wine maker in Burgondy, just bought 20 acres of good soil at Gevrey-Chambertin, planted with red pinot. He has two options, which cannot be changed once they are chosen. By bringing manure from his father'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.

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

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.

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

Pivoting on x_1 and x_2 yields

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

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 Mazis Chambertin?

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 [16] 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	$s = s \ \& \ P$	$\rightarrow$	P	
Decompose	$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \ \& \ P$	$\rightarrow$	$s_1 = t_1 \ \& \ \ldots \ \& \ s_n = t_n \ \& \ P$	
Conflict	$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_n) \ \& \ P$	$\rightarrow$	$\bot$	
		if	$f \neq g$	
Check	$x = s \ \& \ P$	$\rightarrow$	$\bot$	
		if	x occurs in the non-variable term s	
Eliminate	$x = s \ \& \ P$	$\rightarrow$	$x = s \ \& \ P\{x \mapsto s\}$	
		if	x does not occur in s and occurs in P	
Swap	$s = x \ \& \ P$	$\rightarrow$	$x = s \ \& \ P$	
		if	x is a variable, s 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 leftmost place of the constraint. In the last rule, $P\{x \mapsto s\}$ denotes the unification constraint P in which the variable x has been replaced everywhere by the term s. As an example of use, the unification constraint $f(x, g(a, x)) = f(h(a), g(a, y))$ can be transformed as follows:

	$f(x, g(a, x)) = f(h(y), g(a, h(z)))$	
$\rightarrow$	$x = h(y) \ \& \ g(a, x) = g(a, h(z))$	by Decompose
$\rightarrow$	$x = h(y) \ \& \ a = a \ \& \ x = h(z)$	by Decompose
$\rightarrow$	$x = h(y) \ \& \ h(y) = h(z)$	by Eliminate
$\rightarrow$	$x = h(y) \ \& \ y = z$	by Decompose
$\rightarrow$	$x = h(z) \ \& \ y = z$	by Eliminate

which is in solved form. The associated assignment $\{x \mapsto h(z), y \mapsto z\}$ 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 one by an appropriate *specialization*.

Figure 3: Unification constraints

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) will follow the syntax given in the form of the following context free grammar:

$$\mathcal{F} \ := \ \mathcal{X} \text{ or } \mathcal{S} \text{ or } \mathcal{X}(\mathcal{K} \Rightarrow \mathcal{F} \ : \ \mathcal{X}) \text{ or } \mathcal{S}(\mathcal{K} \Rightarrow \mathcal{F} \ : \ \mathcal{X}) \text{ or } \mathcal{X}(\mathcal{K} \Rightarrow \mathcal{F} \ : \ \mathcal{S}) \text{ or } \mathcal{S}(\mathcal{K} \Rightarrow \mathcal{F} \ : \ \mathcal{S})$$

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:

$$X = person(age \Rightarrow Z : nat; eyes \Rightarrow blue : colour; married \Rightarrow Y : person)$$

$$\&$$

$$Y = person(age \Rightarrow Z : nat)$$

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

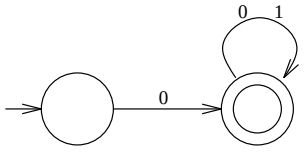
Decompose	$x(Key \Rightarrow y : z) \ = \ x'(Key \Rightarrow y' : z')$	$\rightarrow$	$x = x' \ \& \ y = y' \ \& \ z = z'$
KeyConflict	$x(Key_1 \Rightarrow y : z) \ = \ x'(Key_2 \Rightarrow y' : z')$	$\rightarrow$	$\bot$
		if	Key_1 and Key_2 are different keys
SortConflict	$s = s'$	$\rightarrow$	$\bot$
		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. In this formulation, conflicts between different sort names arise at a later stage. As an example of use:

$$X(age \Rightarrow I : nat) = person(age \Rightarrow J : nat)$$
$$\rightarrow$$
$$X = person \ \& \ I = J$$

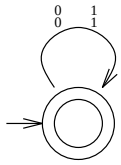
which describe a set X of persons having the same age.

Figure 4: Feature constraints



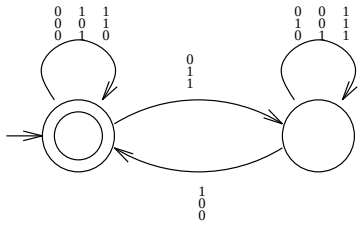
Once we read a 0, then we know that the number (which is written in base 2) is even. Hence we enter a final state marked with a double circle. Then, whatever we read, the number will be even and we stay in that final state.

Figure 5: An automaton accepting even numbers



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 $\binom{0}{1}, \binom{0}{0}, \binom{1}{1}$, we stay in the final state. Since $1 + 1 = 0$ with carry 1, we have the transition by $\binom{1}{0}$ to the state carry. All other transitions are built in the same way.

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

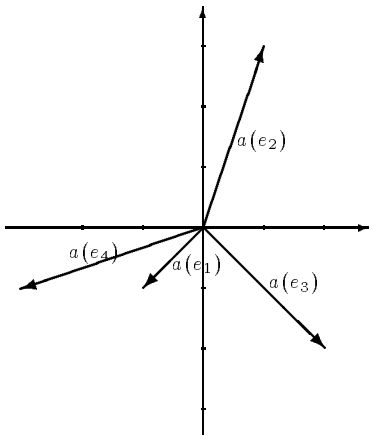


Figure 8: Default vectors

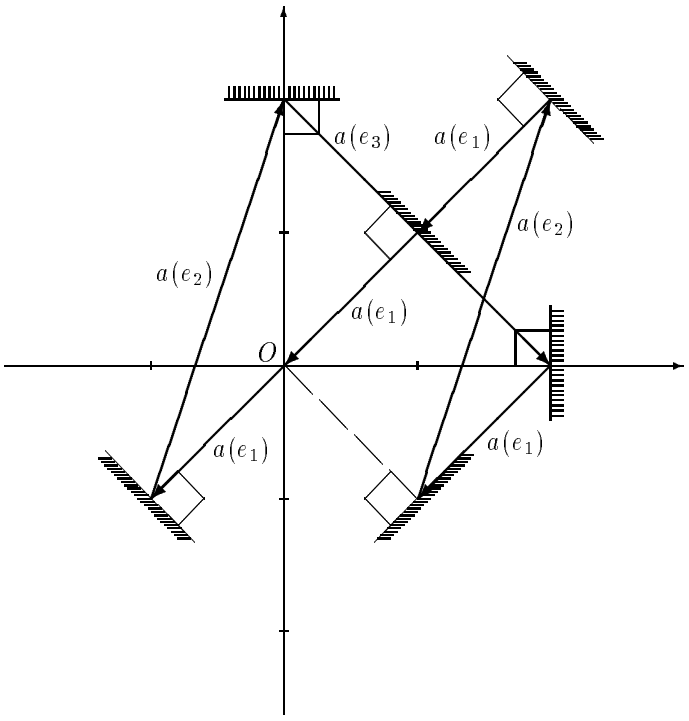


Figure 9: Computation of the solution (4,2,1,0) of the system

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:

$$C := T = T \mid T > T \mid T < T \mid T \geq T \mid T \leq T \mid x \in [a\ b] \mid C\ \&\ C$$

$$T := a \mid x \mid ax \mid T + T \mid T \perp T$$

where x denotes a variable ranging over $\mathbb{N}$, a and b values in $\mathbb{N}$. The atomic constraint $x \in [a\ b]$ is interpreted by $\perp$ 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	$x \in [a\ b]\ \&\ C\ \&\ ?$	$\rightarrow$	$x \in RD(x, C)\ \&\ ?$
		if	$\mathcal{V}ar(C) = \{x\}$
Look Ahead	$x \in [a\ b]\ \&\ C\ \&\ ?$	$\rightarrow$	$x \in RD(x, C)\ \&\ C\ \&\ ?$
		if	$x \in \mathcal{V}ar(C)$, and $RD(x, C) \neq [a\ b]$
Eliminate	$x = a\ \&\ ?$	$\rightarrow$	$x = a\ \&\ ?\{x \mapsto a\}$
		if	$x \in \mathcal{V}ar(?)$
Falsity	$x \in \emptyset\ \&\ ?$	$\rightarrow$	$\perp$
Enumerate	$x \in [a\ b]\ \&\ ?$	$\rightarrow$	$?\{x \mapsto a\}$ **or** $x \in [a+1\ b]\ \&\ ?$
		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 $ca_2 < db_2 + e$. We can therefore choose for a'_2 the largest natural number m such that $cm < db_2 + e$ (unless it is bigger than a_2). Consider now the constraint

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

$$x \in [2\ 3]\ \&\ y \in [2\ 4]\ \&\ z \in [2\ 5]\ \&\ 4x > y + z\ \&\ 3x < y + z$$

which is in normal form, but unsatisfiable, since $4x > y + z$ in the current context forces x to be at least 3, making $3x < y + z$ false. So, 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*. 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 10: Finite domains

We will use the syntax of features as defined in figure 4. Our sorts (or types) will be the four-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:

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

When clear for the context, we do not recall the constant sorts in a feature term. We now consider the following query:

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

where the letters A, X, Y, Z, U denote variables of the appropriate types. The solutions of this constraints 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:

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

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, the domain of feature terms, the domain of strings, and the domain of natural numbers. We therefore obtain the new entailment constraint:

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

which in turn is decomposed in

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

by using an appropriate string matching algorithm.

Figure 11: Bibliographic search