Constraints and Constraint Solving: An Introduction

Jean-Pierre Jouannaud and Ralf Treinen

Laboratoire de Recherche en Informatique
Université Paris-Sud
F91405 Orsay CEDEX, France
http://www.lri.fr/~jouannau, http://www.lri.fr/~treinen

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

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

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

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

In the following sections we will see that, in reality, things are not that simple. In Section 2 we will investigate a particular family of constraint system, so-called feature constraints, in some depth. It will turn out that complete procedures to compute with constraints may be too costly and that we may have to refine our calculus to accommodate for incomplete constraint handling. Section 3 will present such a refinement for constraint logic programming. In Section 4, finally, we will address the question of how the basic algorithms to compute with constraints can be implemented in the same programming calculus than the one that uses these algorithms.

1 A First Approach to Constraint Based Calculi

1.1 First-Order Logic as a Language

The role of this section is to give the basics of first-order logic viewed as a language for expressing relations over *symbolic data items*.

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

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

Different terms may have common ground instances, that is, the two sets of data items that they describe may have a non-empty intersection. When this is the case, the two terms are said to be *unifiable*. For an example, $f(X, g(Y))$ and $f(g(Z), Z)$ are unifiable, since they have $f(g(g(a)), g(a))$ as a common ground instance. The substitution $\tau = \{X \mapsto g(g(a)), Y \mapsto a, Z \mapsto g(a)\}$ is called a *unifier* of these two terms. Does there exist a term whose set of ground instances is the intersection of the sets of ground instances of two unifiable terms s and t? The answer is affirmative, and the *most general instance* of s and t is obtained by instantiating any of them by their *most general unifier* or *mgu*. In the above example, the mgu is $\sigma = \{X \mapsto g(g(Y)), Z \mapsto g(Y)\}$. It should be clear that any unifier τ is an instance of the mgu, by the substitution $\tau' = \{Y \mapsto a\}$ in our example. This is captured by writing the composition of substitutions in diagrammatic order: $\tau = \sigma\tau'$. Renaming the variables of the most general instance does not change its set of ground instances, we say that the most general instance, and the most general unifier as well, are defined up to *renaming of variables*. It should be clear that we can unify n terms $t_1, \ldots, t_n$ as well. In practice, we often need to unify sequences $\overline{s}$ and $\overline{t}$ of terms, that is, find a most general unifier σ such that $s_1\sigma = t_1\sigma, \ldots, s_n\sigma = t_n\sigma$.

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

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

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

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

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

To know more: More on unification of terms and automata techniques is to be found in the lecture on "Constraint Solving on Terms". See also [JK91] for a modern approach to unification and [CDG+97] on automata techniques. The basic logic notations can be found in many text book covering logical methods, for instance [BN98].

1.2 From Logic Programming to Constrained Logic Programming

Logic Programming (LP) is one of the classic formalisms following the paradigm of *declarative programming*. A *program* is a sequence of *definite* Horn clauses, that is, Horn clauses with a non-empty head. The declarative semantics of a logic program P is the least set M_P of ground atoms such that if $B \leftarrow A_1, \ldots, A_n$ is a ground instance of a program clause of P and if $A_1, \ldots, A_n \in M_P$ then $B \in M_P$. Such a least set does always exist for Horn clauses (but not for arbitrary clauses).

Here is an example of logic program:

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

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

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

$$\text{(Resolution)}\ \frac{P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots P_n(\bar{t}_n)}{Q_1(\bar{r}_1\sigma), \ldots, Q_m(\bar{r}_m\sigma), P_2(\bar{t}_2\sigma), \ldots, P_n(\bar{t}_n\sigma)}$$

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

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

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

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

With this program, the goal $path(a,c,L)$ will succeed with answer substitution $\{L \mapsto s(s(s(o)))\}$. However, computation of the goal $plus(X,s(1),X)$ will not terminate. The problem is that, although searching solutions by enumerating all possible computation paths is complete in the limit, this gives us in general not a complete method for detecting the absence of solutions (not to speak of its terrible inefficiency). What we would like here is to have numbers as a primitive data type in LP.

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

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

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

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

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

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

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

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

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

Semantically, an equation $X = s$ must be understood as the set of values (for the variable X) which are the ground instances of the term s. When making a conjunction $X = s \wedge X = t$, we intersect these two sets of values, and therefore, the set of solutions is the same as that of $X = most - general - instance(s, t)$, provided s and t are unifiable. This is why we need to check for satisfiability in our new rule. This satisfiability test will of course be delegated to the old unification algorithm, which is seen here as an "equation solver" which is put aside. Furthermore, we could allow more freedom in the rule by writing in the conclusion of the rule any equations system that is equivalent to $c \wedge d$. This gives the freedom of whether to leave the constraint $c \wedge d$ as it is, or to replace it by some simpler "solved form" (see the lecture on constraint solving on terms for possible definitions of solved forms).

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

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

The crucial abstraction step is now: We obtain CLP when we replace equation systems of terms by any system of description of data with the same two properties as above.

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

1.3 Constraint Systems (First Version)

We use predicate logic for syntax and semantics of constraints since it provides the following features which are important for most of the constraint-based formalisms:

1. first-order variables to denote values, these variables will be shared with the formalism that uses the constraints (for instance a programming language),
2. conjunction as the primary means to combine constraints, since conjunction of formulas corresponds exactly to the intersection of their respective denotation,
3. existential quantification to express locality of variables. It is existential, in contrast to universal, quantification that matters here since the constraint-based mechanisms that we consider are based on *satisfiability* of descriptions.

Hence, a first definition of a *constraint system* (we will refine this definition in Section 3.1) which is appropriate for CLP could be as follows:

A constraint system C consists of

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

CLP programs and the rule of constrained resolution is thus obtained from the above definition by replacing the word "equation system" by "constraint". Some examples of constraint systems are:

1. *Herbrand*: This is the constraint system used in LP when seen as instance of the CLP scheme. Its language is given by an infinite supply of function symbols and equality as the only predicate. The set of constraints is the set of basic constraints whose atomic constraints are the equality atoms, the structure is the structure of ground terms, where function symbols are just constructors of ground terms and equality is completely syntactic, and the algorithm for testing satisfiability is the unification algorithm (usually attributed to Herbrand).
2. *Herbrand over rational terms*: This is the constraint system introduced by Colmerauer in PROLOG II. It is the same as previously, but the structure is the set of ground atoms built from rational terms. The algorithm for testing satisfiability is due to Huet.

3. *Herbrand with inequations*: This is another extension of Herbrand where constraints are existentially quantified systems of equations and inequations. Decidability of satisfiability of constraints has been shown by Alain Colmerauer. This constraint system was the basis of PROLOG III.
4. *Presburger arithmetic*: The language of Presburger arithmetic consists of a constant 0, a unary function s, and a binary operator $+$. Constraints are the basic constraints, interpreted in the usual way over natural numbers. There is a number of standard methods to decide satisfiability of constraints. The set of constraints can be extended to the full set of first-order formulas. There are many ways to show decidability of the full first-order theory, an early algorithm is due to Presburger (hence the name).
5. *Real arithmetic*: The language of real arithmetic contains constants 0 and 1 and binary operators $+$ and $*$, interpreted in the standard way over real numbers. We can take as set of constraints the set of full first-order formulas over this language. It is a surprising result of Tarski that satisfiability of these constraints is again decidable.

We will meet some more interesting constraint systems during this lecture, and in the other lectures as well.

What happens when we extend Presburger arithmetic with multiplication? It is well-known that satisfiability of systems of polynomial equations over natural numbers is undecidable (this is Hilbert's famous 10th problem shown undecidable by Matijacevič), hence Presburger arithmetic with multiplication, does not form a proper constraint system according to the definition given above. This is of course somewhat disappointing since multiplication is certainly something we would like to have when dealing with natural numbers. We will see later how to deal with cases where constraint solving in the general case is not decidable, or just not feasible.

To know more: See [Coh90] for an introduction to Constraint Logic Programming, and [CDJK99], [JM94] for a survey. Much more about constraint solving will be said in the lectures on "Constraint Solving on Terms" and "Combining Constraint Solving".

1.4 Constraints and Theorem Proving

We demonstrate a first usage of constraints in theorem proving, much more is to be seen in the lecture dedicated to the subject.

Let E be a finite set of equations, for example the classical three equations defining group theory:

$$\left\{\begin{array}{r}(x \times y) \times z = x \times (y \times z)\\ x \times 1 = x\\ x \times x^{-1} = 1\end{array}\right.$$

A classical problem is to decide whether a given equation, for example $(x \times y)^{-1} = y^{-1} \times x^{-1}$ in group theory, is a logical consequence of E. This problem, also known as the *word problem*, has been subject to intensive research. The brute force search for a proof using the replacement of equals by equals, although complete, rarely leads to an actual solution. One of the most successful approaches due to Knuth and Bendix is to use *ordered strategies*, that is to use the equations in one way only as so-called *rewrite rules*. Of course, such a strategy is incomplete in general, but completeness can be restored using a completion mechanism based on the computation of some particular equational consequences called *critical pairs*. One requirement of the original method was the *termination* of the rewrite system: the replacement of equals by equals using the ordered strategies should always end up after a finite number of replacement steps.

In the above example of group theory, it is quite easy to fulfill this termination requirement by choosing carefully the way in which to orient the equations. The situation changes if we consider *commutative groups*, adding the equation $x \times y = y \times x$ to the above system. Now the completion procedure fails because commutativity cannot be oriented in either way without loosing termination. Several solutions have been studied to overcome this problem, the solution that interests us here is called *ordered rewriting*.

The idea is very simple: use every equation in one way or the other, depending on the ordering on *the instances on which it is applied*. For example consider the commutativity axiom and assume a total ordering on terms, e.g compare lexicographically the arguments of $\times$, from left to right. Then if $a > b$, $a \times b$ rewrites to $b \times a$ using $x \times y = y \times x$, but not the other way around, since $a \times b > b \times a$, but $b \times a \not> a \times b$.

The equations that we are trying to prove are (implicitly) universally quantified, that is an equation as $x \times y = y \times x$ should be read as "for all ground terms s and t, $s \times t = t \times s$". Hence, as in the case of (C)LP, terms with variables represent the sets of their respective ground instances. In order to describe ordered rewriting we need a total ordering relation on ground terms, denoted $>$. Now, an equation $l = r$ can be replaced by the two constrained rules $l \to r \mid l > r$ and $r \to l \mid r > l$. Since we required the order to be total, we know that for any ground instance of l and r, one of the two rules applies (except in the case $l = r$ when there is of course no need to rewrite). When we require in addition the ordering to be monotonic, that is to satisfy

$$s > t \quad \text{implies} \quad f(\ldots, s, \ldots) > f(\ldots, t, \ldots)$$

for all function symbols f and ground terms s, t, then the ordering on the rule instances propagates to the rewritten terms. Hence, rewriting always decreases terms in the ordering. Assuming then the ordering to be well-founded implies that rewriting a term always ends up in a *normal form*. This latter property is the key to completeness.

Once the equations are split into ordered rewriting rules as described above, we have to complete the rule system in order to obtain a rewrite system that allows to derive all valid equations $s = t$ by rewriting from s and t until a same term u is found. Without going into the technical details here it suffices to say that the completion procedure searches for so-called *critical pairs*, obtained when two rewrite rules $l \to r \mid l > r$ and $g \to d \mid g > d$ can overlap in the same ground term v. For this to happen, say at the top of v to make it simple, a ground substitution σ must exist such that $v = l\sigma = g\sigma$, $l\sigma > r\sigma$ and $g\sigma > d\sigma$. Starting from v, rewriting can yield the two different terms $r\sigma$ and $d\sigma$, and we have to repair this divergence by adding additional rewrite rules. But since the substitution σ is a solution of the *ordering constraint* $l = g \wedge l > r \wedge g > d$, the divergence can be repaired by adding the equation (called critical pair) $r = d \mid l = g \wedge l > r \wedge g > d$, more precisely, the two rules $r \to d \mid l = g \wedge l > r \wedge g > d \wedge r > d$ and $d \to r \mid l = g \wedge l > r \wedge g > d \wedge d > r$. Since we know that the ordering is total, one of the two rules will rewrite one -we don't know which one, of course- of $(r\sigma, d\sigma)$ into the other. This process of adding critical pairs may terminate with a so-called *canonical* rewrite system, that is one which can prove all valid equations by rewriting, or run forever, in which case any valid equation will be provable by rewriting after some time.

Here, we have seen an application of constraints in theorem proving in which constraints play already the role of *defining* the notion of proof, via the definition of ordered rewriting. Constraints can also be used in a completely different way: to restrict the set of possible inferences. In the above discussion, we have overlapped two rules at the top. In practice, we must actually overlap left-hand sides of rules at arbitrary non-variable subterms. Assume now that we can solve the constraints, and replace the ordered rules $l \to r \mid l > r$ by plain rewrite rules $l\sigma \to r\sigma$, where σ is a kind of mgu for the constraint $l > r$. Then overlapping these new rules does not have the same effect at all than overlapping the constrained one they originate from. In the first case, the overlap may happen *inside* the substitution σ, which is impossible for the constrained rule: the use of constraints allows to avoid overlapping inside substitutions originating from previous inferences. This strategy is called *basic*. It is again complete, although it influences in various ways the whole inference process.

To know more: The reader is referred to the lecture on "Constraints and Theorem Proving" for a more complete exposition. See [CT94,Nie99] for some results on ordering constraints on trees and their usage in automated deduction.

1.5 Constraint Satisfaction Problems

So far we have seen that constraints are useful to define an expressive programming paradigm and as a tool to express strategies in automated deduction. Many important problems already employ constraints in the problem formulation, and the task is to find a solution to these constraints. We demonstrate this with a well known game:

$$\begin{array}{rccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = M & O & N & E & Y & \end{array}$$

Each letter denotes a different number between 0 and 9 such that, when read as an addition, the equation is satisfied.

This problem can be easily formalized in our logical setting, by the following program clauses (we omit the clauses defining *plus*, *mul*, *in* and *neq* predicates, and use $10, 100, 1000, 10000$ as abbreviations for their representation using 0 and *succ*):

```
solution(S,E,N,D,M,O,R,Y) :-
/* X_1 is the value of the first row */
   mul(1000,S,Y_1), mul(100,E,Y_2), mul(10,N,Y_3),
   plus(Y_1,Y_2,Y_4), plus(Y_3,Y_4,Y_5), plus(D,Y_5,X_1),
/* X_2 is the value of the second row */
   mul(1000,M,Z_1), mul(100,O,Z_2), mul(10,R,Z_3),
   plus(Z_1,Z_2,Z_4), plus(Z_3,Z_4,Z_5), plus(E,Z_5,X_2),
/* X_3 is the value of the third row */
   mul(10000,M,T_1), mul(1000,O,T_2), mul(100,N,T_3),
   plus(T_1,T_2,T_5), plus(T_3,T_5,T_6), plus(T_4,T_6,T_7),
   mul(10,E,T_4), plus(Y,T_7,X_3),
/* value of row1 + value of row2 = value of row3 */
   plus (X_1,X_2,X_3),
/* S ... Y between 0 and 9 */
   in(S,0,9), in(E,0,9), ..., in(Y,0,9),
/* S ... Y all different */
   neq(S,E), neq(S,N), ..., neq(R,Y).
```

Prompted by the query $plus(X, Y, Z)$, resolution will search for the values of all the variables. This brute force approach consists in generating all $10 * 9 * \ldots * 4 = 604.800$ assignments of different digits to the letters and checking whether they satisfy the clauses.

We can of course skip this logical formulation, and formulate the problem directly as a constraint satisfaction problem of integer arithmetic, using appropriate predicates and function symbols:

$$
\begin{aligned}
&1000 * S + 100 * E + 10 * N + D + 1000 * M + 100 * O + 10 * R + E \\
&\quad = 10000 * M + 1000 * O + 100 * N + 10 * E + Y \\
&\wedge S \in [0,9] \wedge E \in [0,9) \wedge N \in [0,9) \wedge D \in [0,9) \\
&\wedge M \in [0,9) \wedge O \in [0,9) \wedge R \in [0,9) \wedge Y \in [0,9) \\
&\wedge S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y
\end{aligned}
$$

This formulation is much more concise and elegant, of course. Again, we can solve the constraint by a brute force enumeration of the values of the variables, which would yield about the same amount of work as previously. Modern constraint programming systems, however, can solve this problem in a fraction of a second using a much more clever approach based on *constraint propagation*. Constraint propagation implements a kind of meta-reasoning on the constraints themselves, in order to cut down the search. For an example, M must be equal to 1, since it is a carry on. In turn, this implies that S must be equal to 8 or 9. Constraint propagation on intervals (also called *finite domains*) was first implemented in the CHIP system [DSH90].

Although the above discussion is certainly just a toy example, this kind of application is of paramount importance in practice. Problems of this kind occur, for instance, in all kind of scheduling problems (think of making a staff plan under restricted availability of the persons, scheduling task among different machines where the execution of tasks has to obey to a certain order, making a timetable for a school, etc.). A more detailed account of this approach is given in Section 4.4.

In the above example, the constraint domain, a finite subset of the set of integers, was quite familiar. Integers are readily available in every programming language, addition and multiplication pre-exist as well, making it easy to design and program the constraint solving algorithm. In other problems, the key is to *formulate* the appropriate constraint system itself, and to find the associated constraint solving algorithm. Such new constraint systems usually emerge in research problems. For example, feature constraints described in Section 2 originate from natural language processing applications. Since then, they have been found very useful in various areas, ranging from object-oriented programming to heterogeneous data bases. Terms with integer exponents [Com95] were first introduced by Hsiang for extending LP, before to find new applications in second-order unification. And actually, the domain of infinite rational trees itself can be seen as a constraint system, since every such tree is the solution of a set of equations with occur-check. For example, the equation $X = g(a, X)$ describes the infinite rational tree $g(a, g(a, g(a, \ldots g(a, \ldots))))$, in the sense that it is the unique solution of the equation. Of course, certain sets of equations with occur-check, like $X = g(Y, X) \wedge Y = h(Z)$ describe sets of rational terms, here all terms which are an instance of $g(h(Z), g(h(Z), g(h(Z), \ldots g(h(Z), \ldots))))$. Again, it is important to realize that it is easier to express problems and compute with these constraints seen as descriptions, than with their denotations.

To know more: More about this kind of application is to be told in the lecture on "Building Industrial Applications with Constraint Solving".

1.6 Constraint Based Grammar Formalisms

So called *unification grammars* have a long history in computational linguistics where they are employed for the analysis and generation of natural language. Originally, these grammars were defined without using constraints or even predicate logic. The name "unification grammar" is a bit unfortunate since "unification" is the operation of resolving equations, calling them *constraint grammars* is much more appropriate since this name emphasizes the declarative aspect of the formalism. However, these constraint grammars are usually not considered as a generic formalism that can be used with any constraint system (as it is the case with constraint logic programming) but come with so-called feature constraints that we will see in an example below, and that we will investigate in greater depth in Section 2.

We will demonstrate the principle with a very simple example. Consider a context-free grammar for English which contains the following rules:

$$\begin{aligned} S &\to NP\ VP \\ NP &\to D\ N \\ VP &\to V\ NP \end{aligned}$$

where the non-terminals stand for *Subject*, *Noun Phrase*, *Verb Phrase*, *Determiner*, and *Noun* respectively. Additionally, there are so-called lexical rules that represent a dictionary:

$$\begin{aligned} N &\to \texttt{girl} \\ N &\to \texttt{movie} \\ N &\to \texttt{movies} \\ D &\to \texttt{a} \\ D &\to \texttt{the} \\ V &\to \texttt{watches} \end{aligned}$$

This grammars licenses correct English sentences like

`the girl watches a movie`

but also incorrect sentences like

`the girl watches a movies`

The problem is that context-free grammar rules can only define the possible skeletons of a phrase, however, there are additional conditions to be met for a sentence to be grammatically correct, as for instance correspondence of the numeros of the subject and the verb.

Hence, we augment our context-free grammar with constraints. While a context-free grammar describes a set of parse trees, a constraint grammar describes a set of attributed parse trees. In such an attributed parse tree, nodes can be decorated with functional attributes called *features*, and the constraints in the grammar impose relations between the attributes of the nodes for an attributed parse tree to be admissible, that is to describe a correct English sentence. For example, we might express the correspondence of the numeros of subject and verb by adding some constraints to the structural and to the lexical grammar:

$$\begin{aligned} S &\to NP\ VP \mid NP.numeros = VP.numeros \\ NP &\to D\ N \mid NP.numeros = D.numeros \land D.numeros = N.numeros \\ VP &\to V\ NP \mid VP.numeros = V.numeros \\ N &\to \texttt{girl} \mid N.numeros = singular \\ N &\to \texttt{movie} \mid N.numeros = singular \end{aligned}$$

$N \rightarrow$ `movies` | $N.numeros = plural$
$D \rightarrow$ `a` | $D.numeros = singular$
$D \rightarrow$ `the`
$V \rightarrow$ `watches` | $V.numeros = singular$

Now, the constraints only allow sentences where the numeros of the noun phrase coincides with the numeros of the verb phrase and where in a noun phrase the numeros of the determiner coincides with the numeros of the noun. An interesting point to note about this grammar is that constraints are used to accomplish two different tasks:

- to express restrictions on the attributes of components, for instance that in a noun phrase the numeros of determiner and noun must coincide; and
- to define the attributes of nodes, for instance the numeros of a noun phrase is defined to be the numeros of its determiner.

Also, note that we did not specify the numeros of the determiner `the` which might be singular or plural. Hence, our grammar licenses both noun phrases `the movie` and `the movies`.

To know more: This section follows the exposition of constraint grammars in [Smo92]. See [Kel93] for the use of feature constraints in computational linguistics.

1.7 Constraint-Based Program Analysis

Constraint based program analysis stands a bit apart from the constraint-based formalisms mentioned before since here constraints are not used as a description language to augment the descriptive power of another formalism (programming language, deduction calculus, grammar) but are rather used as a means to *approximate* the meaning of a program.

Constraint-based program analysis uses *set constraints* as a language to approximate the meaning of a program. Different systems of set constraints have been investigated during the last decade, here we use the constraint system of [HJ90].

The logical language of our set constraint system contains a signature Σ, a binary union operator, and a constant $\top$. In addition, the language contains for every n-ary function symbol $f \in \Sigma$ so called *projections* $f_1^{-1}, \ldots, f_n^{-1}$ which are all unary function symbols. The only predicate symbol is the binary symbol $=$. Constraints are possibly existentially quantified conjunctions of atomic constraints.

Basic data items in the interpretation of set constraints are *sets of ground terms*. Function symbols operate now on sets, that is the meaning of $f(M, N)$ is

$\{f(m,n) \mid m \in M, n \in N\}$. The symbol $\top$ denotes the set of all ground terms, $\cup$ and $=$ have their usual meaning, and the meaning of $f_i^{-1}(M)$ is

$$\{m_i \mid \exists m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_n \; f(m_1, \ldots, m_i, \ldots, m_n) \in M\}$$

Satisfiability of these constraints is decidable but non-trivial, see the literature for pointers.

As a simple example we demonstrate the approximation of the meaning of a logic program; the method works as well for functional or imperative programs. Consider the well-known logical program for the concatenation of lists:

```
conc(nil,V,V).
conc(H:X,Y,H:Z) ← conc(X,Y,Z).
```

No variable is shared among clauses, this is necessary for the abstraction to work (and can easily be achieved, the case given, by renaming of variables). The abstraction uses one set variable Γ, corresponding to the predicate *conc*, and variables V, H, Y, Z. First, we state that a successful computation of *conc* can be obtained by starting from the first clause or by starting from the second clause:

$$\Gamma = conc(nil, V, V) \cup conc(H : X, Y, H : Z)$$

To get constraints for the variables V, X, Y, Z, H we have to look at the clauses where they occur (note that every of these variables has to occur in exactly one clause). The variables V and H do not occur in the bodies of their clause, hence there is no restriction on them and they can denote any term:

$$V = \top$$
$$H = \top$$

The variables X, Y and Z can assume any value that stems from *some* successful computation of *conc*. More precisely, X can take any value such that for *some value* of Y and Z the computation of $conc(X, Y, Z)$ is successful. This is were the approximation happens: the point-wise dependencies between variables are ignored. We get the constraints

$$X = conc_1^{-1}(\Gamma)$$
$$Y = conc_2^{-1}(\Gamma)$$
$$Z = conc_3^{-1}(\Gamma)$$

Hence, the complete set constraint to resolve is

$$\Gamma = conc(nil, V, V) \cup conc(H : X, Y, H : Z) \wedge V = \top \wedge H = \top \wedge$$
$$X = conc_1^{-1}(\Gamma) \wedge Y = conc_2^{-1}(\Gamma) \wedge Z = conc_3^{-1}(\Gamma)$$

This constraint is satisfiable, its minimal solution assigns to V, H, Y and Z the set of all ground terms, and to X the set of all lists.

To know more: A recent survey and bibliography of constraint-based program analysis can be found in [PP97].

2 A Case Study of a Constraint System: Feature Constraints

2.1 Motivation

The constraint system Herbrand, that is first-order terms, is often used to model symbolic data. However, this constraint system is for many applications too inflexible. This inflexibility has two sources, one is the use of classical first-order trees as underlying data model, and the other is the particular set of atomic constraints provided by the Herbrand constraint system.

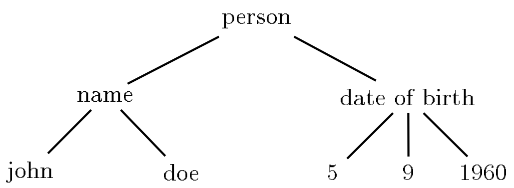


Fig. 1. Classical terms to represent data.

The problem with using classical trees as a data model is that the subtrees of a tree described in the Herbrand constraint system are numbered, that is we always have to remember the meaning that we have in mind with the first, second, third, ... subtree of a tree. In the example of Figure 1 we have to remember that in the subtree labeled *name* the first component is the first name and the second component is the family name, and that the date of birth is stored in the format day-month-year and not, for instance, month-date-year. In many applications the use of symbolic names to denote subtrees is much more natural. This is illustrated in Figure 2 where we used a tree with labeled edges, a so-called *feature tree*, to represent some data about a movie (an exact definition of feature trees will follow soon). Using arbitrary names to denote components recalls of course the notion of *record* which is known from many programming languages.

The second problem with the Herbrand constraint system comes from its choice of atomic constraints. If we write a constraint $x = f(y, z)$ then we are expressing several things at the same time:

1. that the root of the tree x is labeled with the symbol f,
2. that there are exactly two subtrees at the root of the tree f,
3. that the first subtree of x is denoted by y, and that the second subtree of x is denoted by z.

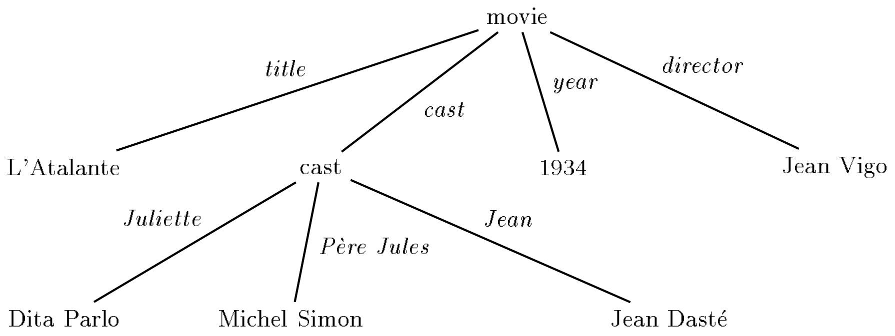


Fig. 2. Feature trees to represent data.

That the root of x is labeled by f is easily expressed in Herbrand by using an existential quantifier:

$$\exists y, z\; x = f(y, z)$$

assuming that f is known to be binary. The other properties, however, are much more cumbersome to express in Herbrand since they require the use of *disjunctions*:

$$\bigvee_{f \in \Sigma_2} \exists y, z\; x = f(y, z) \quad x \text{ has two subtrees}$$

$$\bigvee_{f \in \Sigma_2} x = f(y, z) \quad \text{the first (second) subtree of } x \text{ is } y\ (z)$$

where Σ_2 is the set of binary function symbols. Note that disjunctions are not available in the Herbrand constraint system, hence we either have to extend the Herbrand system accordingly, or we have to accommodate for disjunctions in the programming formalism. Even worse, the signature might not be known in advance since we want to have the freedom to use any function symbol in the course of computation (imagine a system where the user interactively enters data items, using any function symbols he likes). Not knowing the signature in advance is usually modeled by assuming the signature to be infinite, for instance to be the set of all words formed over some alphabet. In this case the above disjunctions are not even formulas of first-order predicate logic.

This is where *feature constraints* come in: Atomic feature constraints are more fine-grained than Herbrand-equations. In the constraint system CFT, for example, there are exactly the three kinds of atomic constraints that correspond to the three properties listed above: *label constraints* Ax (sometimes called *sort* *constraints*) to specify the symbol at the root node of a tree, *arity constraints* $x\{f_1, \ldots, f_n\}$ to specify the labels of the edges departing from the root, and *subtree* constraints $x[f]y$ to specify the subtree of a tree at a specific edge.

2.2 The constraint system CFT

Formal Definition We assume an infinite set $\mathcal{F}$ of *features* ranged over by f, g and a set $\mathcal{L}$ of *labels* ranged over by a, b.

A *path* π is a word of features. The *empty path* is denoted by ϵ and the free-monoid concatenation of paths π and π' as $\pi\pi'$. A path π' is called a *prefix of* π if $\pi = \pi'\pi''$ for some path π''. A *tree domain* is a non-empty prefix closed set of paths.

A *feature tree* τ is a pair $(D,\, L)$ consisting of a tree domain D and a function $L : D \to \mathcal{L}$ that we call *labeling function* of τ. Given a feature tree τ, we write D_τ for its tree domain and L_τ for its labeling function. For instance, $\tau_0 = (\{\epsilon, f\},\, \{(f, a)\})$ is a feature tree with domain $D_{\tau_0} = \{\epsilon, f\}$ and $L_{\tau_0} = \{(f,\, a)\}$. A feature tree is *finite* if its tree domain is finite, and *infinite* otherwise. The *root* of τ is the node ε. The *root label* of τ is $L_\tau(\varepsilon)$, and $f \in \mathcal{F}$ is a *root feature* of τ if $f \in D_\tau$. Given a tree τ and a root feature f of τ, we write as $\tau[f]$ the subtree of τ at path f; formally $D_{\tau[f]} = \{\pi' \mid f\pi' \in D_\tau\}$ and $L_{\tau[f]} = \{(\pi',\, a) \mid (f\pi',\, a) \in L_\tau\}$.

An *atomic CFT constraint* is one of the following forms:

$$a(x),\; x[f]y,\; x\{f_1, \ldots, f_n\},\; x = y$$

where $a \in \mathcal{L}$ and $f, f_1, \ldots, f_n \in \mathcal{F}$. A CFT constraint is a possibly existentially quantified conjunction of atomic CFT constraints.

We define the structure CFT over feature trees in which we interpret CFT constraints. Its universe consists of the set of all feature trees. The constraints are interpreted as follows:

$$\begin{aligned} \tau_1[f]\tau_2 &\text{ iff } \tau_2 = \tau_1[f] \\ a(\tau) &\text{ iff } (\varepsilon,\, a) \in L_\tau \\ \tau F &\text{ iff } F = D_\tau \cap \mathcal{F} \end{aligned}$$

and $=$ is of course equality. Hence, $\tau_1[f]\tau_2$ means that τ_2 is the subtree of τ_1 at feature f, $a(\tau)$ means that a is the root label of τ, and τF means that F is the set of root features of τ.

Satisfiability of CFT Constraints Since existential quantifiers are not relevant when testing satisfiability of constraints we can simply ignore them for the moment, hence for the purpose of checking satisfiability of constraints we assume that a constraint is just a conjunction of atomic constraints.

A CFT constraint c is said to be a *solved form* if it contains no equation and

1. if $x[f]y \in c$ and $x[f]z \in c$ then $y = z$

2. if $A(x) \in c$ and $B(x) \in c$ then $A = B$
3. if $xF \in c$ and $x[f]y \in c$ then $f \in F$
4. if $xF \in c$ and $xG \in c$ then $F = G$

A variable x is called *constrained* by a solved form c if c contains an atomic constraint of one of the forms $A(x)$, xF of $x[f]y$. The set of constrained variables of c is written $C(c)$. For example,

$$A(x) \wedge x[f]y \wedge x[g]z \wedge y\{h, k\} \wedge y[h]x$$

is a solved form, its constrained variables are x and y. The crucial property of solved forms is that

Proposition 1. *For every solved form c we have that*

$$CFT \models \forall \bar{x}\, \exists \bar{y}\; c$$

where $\bar{x} = V(c) - C(c)$ and $\bar{y} = C(c)$

Hence, in particular, every solved form is satisfiable. In continuation of the above example we get that the following formula is valid:

$$\forall z\, \exists x, y \Big(A(x) \wedge x[f]y \wedge x[g]z \wedge y\{h, k\} \wedge y[h]x \Big)$$

Proof. (Sketch) Beware that the solved form may contain cycles between constrained variables, as between x and y in the above example. Given feature trees as values of the non-constrained variables, we can see a solved form as a function mapping a tuple of feature trees (indexed by the constrained variables) into a tuple of feature trees of the same kind. We get a solution of the solved form as the minimal fixed point of this functional (we can assume w.l.o.g. that there is a label constraint for every constrained variable). This fixed point is obtained by starting with a tuple of trees containing just a root node plus the subtrees corresponding to the values of the unconstrained variables, and taking the least upper bound of the iterative application of the function.

We describe now a *rule-based* algorithm for deciding satisfiability of CFT constraints. Such a rule-based algorithm is given by a set of rules transforming constraints into constraints. This yields a decision algorithm for satisfiability provided that the three following properties hold:

1. Soundness: Application of the rules preserves semantics.
2. Termination: There is no infinite sequence of rule applications.
3. Completeness: If no rules applies then we have reached a special form from which we can immediately tell whether it is satisfiable or not.

Note that confluence of the calculus is not necessary here since we require preservation of semantics, hence different normal forms of the same constraint are logically equivalent.

$$\text{(Eq)} \quad \frac{x = y \wedge \phi}{x = y \wedge \phi[y/x]} \quad x \text{ occurs in } \phi,\ x \neq y$$

$$\text{(Triv)} \quad \frac{x = x \wedge \phi}{\phi}$$

$$\text{(SClash)} \quad \frac{Ax \wedge Bx \wedge \phi}{\bot} \quad A \neq B$$

$$\text{(Fun)} \quad \frac{x[f]y \wedge x[f]z \wedge \phi}{x[f]z \wedge y = z \wedge \phi}$$

$$\text{(AClash)} \quad \frac{xF \wedge xG \wedge \phi}{\bot} \quad F \neq G$$

$$\text{(FClash)} \quad \frac{xF \wedge x[f]y \wedge \phi}{\bot} \quad f \notin F$$

Fig. 3. Simplification rules for CFT.

An equation $x = y$ is *eliminated* in a constraint ϕ if $x = y$ is an atomic constraint of ϕ (y is a variable different from x) and x occurs only once in ϕ.

The rules are given in Figure 3. We write $c \rightarrow_{CFT} d$ if the constraint c reduces in one step by the system of Figure 3 to the constraint d.

Here is an example of a reduction to solved form. The redeces of the rule applications are underlined.

$$\begin{array}{l}
\underline{x = y} \wedge x[f]x' \wedge y[f]y' \wedge Ax' \wedge By' \\
x = y \wedge \underline{y[f]x'} \wedge \underline{y[f]y'} \wedge Ax' \wedge By' \\
x = y \wedge \underline{x' = y'} \wedge y[f]y' \wedge Ax' \wedge By' \\
x = y \wedge x' = y' \wedge y[f]y' \wedge \underline{Ay'} \wedge \underline{By'} \\
\bot
\end{array}$$

Proposition 2. *If $c \rightarrow^*_{CFT} d$ then $CFT \models c \Leftrightarrow d$.*

Proof. This is immediate by definition of the semantics of CFT.

Proposition 3. *The system of Figure 3 is terminating.*

Proof. We show this using a lexicographic combination of measuring function ψ_1, $\psi_2, \ldots$, each mapping a constraint to some natural number. To show termination of the system it is sufficient to show that if c reduces to d then there is some i such that ψ_1 to ψ_{i-1} are unchanged and ψ_i strictly decreases.

More precisely, we define three measuring functions:

- $\psi_1(\phi)$ is the number of subtree constraints of ϕ,

– $\psi_2(\phi)$ is the number of atomic constraints of ϕ,
– $\psi_3(\phi)$ is the number of variables of ϕ that are *not* eliminated in ϕ.

The termination argument is now given by the following table:

Rule	ψ_1	ψ_2	ψ_3
Eq	$=$	$=$	$<$
Triv	$=$	$<$	
Fun	$<$		
{S,A,F}Clash	$\leq$	$<$	

Proposition 4. *Any constraint which is not reducible by $\rightarrow_{CFT}$ is either $\perp$ or of the form $e \wedge c$ where c is a solved form and e is a system of equations $x_1 = y_1 \wedge \ldots \wedge x_n = y_n$ where all x_i are eliminated in $e \wedge c$.*

Theorem 5. *Satisfiability of CFT-constraints is decidable in quasi-linear time.*

Proof. Given a constraint c, we apply the transformation rules of Figure 3 in any order until we obtain a constraint d that is no longer reducible. This process terminates by Proposition 3. By Proposition 4, d is either $\perp$ and hence not satisfiable, or of the form $e \wedge c$, where c is a solved form and e is eliminated, and hence satisfiable. By Proposition 2, c is satisfiable iff d is.

It is quite easy to see that the number of possible reductions is linear in the number of atomic constraints of c, hence a normal form of c can be found in polynomial time. An algorithm that exhibits quasi-linear complexity can be found in the literature.

Note that this complexity result is about the so-called off-line complexity, that is it applies to the situation were the input is the complete constraint. In case of the CLP calculus presented in Section 1.2, however, the constraint has to be checked for satisfiability at each stage of the computation. In this way, even a linear complexity for every single step sums up to quadratic complexity when we consider the totality of all satisfiability checks. Although this is so far not in the calculus (it will be realized in the calculus of Section 3), it is straightforward to extend the programming calculus in such a way that the constraint is stored in a logically equivalent *simplified* form. Then it makes sense to ask for the on-line complexity of the problem: Given a sequence of atomic constraints, $a_1, \ldots, a_n$, what is the complexity of determining the smallest i such that $a_1, \ldots, a_i$ is unsatisfiable? An algorithm solving this problem can store the constraints seen so far in a simplified form. The abstract machine presented in the paper, based on this principle, has a quasi-linear complexity even in this on-line scenario.

Negated Constraints and Independence We might now try to add negated constraints to the constraint system CFT, thus considering constraints like

$$Ax \wedge x[f]y \wedge \neg\exists z\Big(x[g]z \wedge Bz\Big) \wedge \neg\exists v\Big(x[g]v \wedge v\{f, g, h\}\Big)$$

Negative constraints provide additional expressivity. There is another important motivation, stemming from the entailment problem considered below, which we will encounter in Section 4.

The treatment of conjunctions of positive and negative constraints is simplified by an important property of CFT: The independence of negative constraints (or short: independence).

A constraint system is said to be *independent* if for all constraints $\gamma, \gamma_1, \ldots, \gamma_n$ (remember that we require the set of constraints to be closed under existential quantification) the following holds:

$$\gamma \wedge \neg\gamma_1 \wedge \ldots \wedge \neg\gamma_n \text{ is satisfiable} \qquad \Leftrightarrow \qquad \text{for all } i\text{: } \gamma \wedge \neg\gamma_i \text{ is satisfiable}$$

This is equivalent to say that

$$\gamma \models \gamma_1 \vee \ldots \vee \gamma_n \qquad \Leftrightarrow \qquad \text{there is an } i \text{ such that } \gamma \models \gamma_i$$

Hence, independence allows us to reduce the problem of checking that a conjunction of positive and negative constraints $\gamma \wedge \neg\gamma_1 \ldots \wedge \neg\gamma_n$ is satisfiable, to the problem of checking that for all indices i the entailment judgment $\gamma \models \gamma_i$ does *not* hold.

An example of a constraint system that is not independent is (Presburger or real) arithmetic with multiplication: $x * y = 0 \models x = 0 \vee y = 0$, but $x * y = 0$ does not imply any of $x = 0$ or $y = 0$.

CFT enjoys the independence property. This is the case only because we required the set of label symbols and the set of feature symbols to be infinite. If the set of labels is finite, say $\{A_1, \ldots, A_n\}$, then independence does not hold since $\mathit{true} \models (A_1 x \vee \ldots \vee A_n x)$ but true implies no particular $A_i x$. A similar counterexample can be found if the set of features is finite.

Constraint Entailment Hence, in order to cope with positive and negative constraints we have to solve the so-called *entailment* problem: Given constraints γ and γ', does $\gamma \models \gamma'$ hold?. Besides its use for testing satisfiability of positive and negative constraints, entailment is of paramount importance in the paradigm of constraint programming (see Section 4).

The first observation is that, in an entailment problem $\gamma \models \gamma'$, we can assume γ to contain no existential quantifier. This is justified by the following equivalences, where we can assume without loss of generality that the set of local variables of γ, $\bar{x}$, is disjoint with $V(\gamma')$ (otherwise we can rename the local variables of γ since CFT, as all constraint systems, is closed under renaming of bound variables):

$$\begin{aligned} & (\exists \bar{x}(a_1 \wedge \ldots \wedge a_n)) \models \gamma' \text{ is valid} \\ \Leftrightarrow\ & \forall \bar{x}((a_1 \wedge \ldots \wedge a_n) \Rightarrow \gamma') \text{ is valid} \\ \Leftrightarrow\ & (a_1 \wedge \ldots \wedge a_n) \models \gamma' \text{ is valid} \end{aligned}$$

The existential quantifier on the right-hand side of the entailment problem, however, is essential and can not be dropped: $x\{f\} \models \exists y\, x[f]y$ is a valid entailment judgment but $x\{f\} \models x[f]y$ is not valid.

We will demonstrate the test of entailment of CFT constraints with some examples. The exact criterion can be found in the literature. The judgment

$$x[f]y \wedge Ay \wedge x[f]z \wedge Bz \models \exists v\, x[g]v$$

holds since the left-hand side is not satisfiable, the rules of Figure 3 reduce the left-hand side to $\perp$. Hence the entailment judgment is vacuously true. The judgment

$$x[f]y \wedge Ay \models \exists z(x[f]z \wedge Bz)$$

does *not* hold since the left-hand side is satisfiable, but both sides together are not. Hence, there is a valuation that makes the left-hand side true and the right-hand side false, that is, implication does not hold. The judgment

$$x[f]y \wedge Ay \wedge x[g]z \wedge Bz \models \exists v(x[f]v \wedge x[g]z)$$

holds since the information on the right-hand side is directly contained in the left-hand side after proper instantiation of the local variable v. The judgment

$$x\{f,g\} \wedge Ax \wedge x[f]z \wedge y\{f,g\} \wedge Ay \wedge y[f]z \models x = y$$

does *not* hold since the values of x and y might differ at the feature g. The judgment

$$x\{f,g\} \wedge Ax \wedge x[f]x \wedge x[g]y \wedge y\{f,g\} \wedge Ay \wedge y[f]x \wedge y[g]y \models x = y$$

does hold. This case needs a more involved reasoning than the preceding cases:

We say that a variable x is *determined* by a constraint if the constraint contains a label constraint Ax, an arity constraint $x\{f_1,\ldots,f_n\}$ and selection constraints $x[f_i]y_i$ for each $i = 1,\ldots,n$. In the above example, x and y are both determined. The left-hand side does not contain any other variable, the consequence is that the left-hand-side has *exactly one solution*.

Furthermore, if we identify in the left-hand side x with y then we don't get a contradiction with the rules of Figure 3. Hence, there is a common solution of both sides. Since the left-hand side allows only one solution, we conclude that *all* solutions of the left-hand side also satisfy the right-hand side, that is that entailment holds. Furthermore, the judgment

$$x\{f,g\} \wedge Ax \wedge x[f]x \wedge x[g]z \wedge y\{f,g\} \wedge Ay \wedge y[f]x \wedge y[g]z \models x = y$$

also holds. This is a generalization of the preceding case since we now have a non-determined variable (that is, z) on the left-hand side. The argument is similar to the preceding case, except that now the left-hand side has for every value of z exactly one solution in the determined variables x and y, and that the left-hand side under the identification $x = y$ is satisfiable for any value of z.

The exact result is

Theorem 6. *Entailment of CFT constraints can be checked in quasi-linear time.*

To know more: A much more detailed exposition with an exact statement of the entailment criterion, proofs and the description of an abstract machine for testing satisfiability and entailment of CFT constraints can be found in [ST94].

2.3 First-class features

The feature constraint system CFT presented above has one serious drawback: feature symbols are fixed symbols. Variables denote feature trees but not feature symbols! It would be nice if variables could also denote feature symbols, that is if feature symbols would be given *first-class status*. This would allow much more expressivity, for instance we could just *program* an iteration over all root features of a feature tree.

We thus define the feature constraint system F: The logical language now contains two sorts: features and trees. Feature symbols are now constants of sort feature, and subtree selection $x[v]y$ is now a ternary predicate symbol of profile (trees, features, trees). Label and Arity constraints are as in CFT, and there is an equality predicate on both sorts. Here is an example of an F-constraint:

$$x\{f,g\} \wedge x[v]y \wedge v = f \wedge A(y)$$

Theorem 7. *Satisfiability of F-constraints is NP-complete.*

Proof. It is obvious that satisfiability of constraints can be checked in NP: If there is an atomic constraint $x[v]y$ and an arity constraint xF then we guess the value of v among the features in F; if there is no arity constraint for x then it is safe to take for v any new feature symbol.

To show that the problem is indeed NP-hard we encode 3-SAT in the satisfiability problem of F-constraints. Let $F = L_1 \wedge \ldots \wedge L_k$ be a propositional formula in conjunctive normal form, that is every L_i is a disjunction of three literals. We assume that the propositional variables are taken from the set $\{x_1, \ldots, x_n\}$. We construct an F-constraint ϕ_F such that F is satisfiable if and only if the constraints ϕ_F is satisfiable. The construction of F uses the variables $x_1^+, \ldots, x_n^+, x_1^-, \ldots, x_n^-, y_1, \ldots, y_k$. The formula ϕ_F is the conjunction of three subformulas $\phi_F^1 \wedge \phi_F^2 \wedge \phi_F^3$ that we will now describe:

1. For every i, either x_i^+ has root feature $True$ and x_i^- has root feature $False$, or the other way round.

$$\phi_F^1 = \bigwedge_{i=1}^{n} \exists z \Big(z\{p,n\} \wedge z[p]x_i^+ \wedge z[n]x_i^- \wedge \\ \exists i, v(z[i]v \wedge True(v)) \wedge \exists i, v(z[i]v \wedge False(v)) \Big)$$

2. Encoding of the clause L_i:

$$\phi_F^2 = \bigwedge_{i=1}^{k} \Big(y_i\{1,2,3\} \wedge y[1]q_1 \wedge y[2]q_2 \wedge y[3]q_3 \Big)$$

where L_i is $p_1 \vee p_2 \vee p_3$, and

$$q_j = \begin{cases} x_i^+ \text{ if } p_j = x_i \\ x_i^- \text{ if } p_j = \neg x_i \end{cases}$$

To give an example of this construction: If

$$F = \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3)}_{L_1} \wedge \underbrace{(\neg x_2 \vee x_4 \vee \neg x_5)}_{L_2}$$

then we get

$$\begin{aligned} \phi_F^2 = {} & y_1\{1,2,3\} \wedge y_1[1]x_1^+ \wedge y_1[2]x_2^- \wedge y_1[3]y_3^- \wedge \\ & y_2\{1,2,3\} \wedge y_2[1]x_2^- \wedge y_2[2]x_4^+ \wedge y_2[3]x_5^- \end{aligned}$$

3. All clauses L_i evaluate to true:

$$\phi_F^3 = \bigwedge_{i=1}^{k} \exists j, v(y_i[j]v \wedge True(v))$$

Does NP-completeness of the satisfiability problem imply that the constraint system is useless? There are two answers to this question:

1. It depends on the application. In applications like automatic theorem proving even NP might be an acceptable complexity since the search space is so huge that any means to restrict search by constraints is welcome. In programming, however, NP complete constraint solving is much harder to accept. One can argue that the complexity of a program should stem from the algorithm, as expressed in the program, and not from the basic mechanisms that handles the data structures. Otherwise, it will be very difficult to obtain a useful complexity model. In other words, one should try to keep built-in constraint solving in programming simple (say linear, or at least polynomial of small degree), and have means to have more complex constraint solving programmed by the user.
2. The constraints might be hard to solve in the general case, however, they are easy to resolve once we have sufficient information about the values of the variables. In the case of F: Solving a constraint where all the feature variables are instantiated is nothing but CFT constraint solving, and this we know to do very efficiently. So one could attempt to modify our constraint-based formalism and allow constraints not to be solved immediately, that is to allow incomplete constraint solving methods.
   In the next section we will pursue this approach.

To know more: The constraint-system with first-class features has been introduced in [Tre93].

3 Programming with Incomplete Constraint Solvers

We will now refine the model of CLP presented in Section 1.2 to accommodate for incomplete constraint solving as discussed at the end of the preceding section. Furthermore, we would like to be more exact with the process of *constraint simplification* for reasons discussed after the proof of Theorem 5.

3.1 Constraint Systems (Second Version)

First we will refine our definition of a constraint system: A constraint system consists of

1. a language L of first-order logic,
2. a subset C of the set of first-order formulae of L , called the set of *constraints* of C, containing all atomic constraints, and closed under conjunction, existential quantification and renaming of bound variables,
3. a first-order structure A interpreting L,
4. a subset $S \subseteq C$ of *solved constraints* such that all constraints of S are satisfiable in A,
5. a computable partial *inference function*

$$I: S \times C \rightsquigarrow \bot \cup (S \times P(C))$$

   such that
   (a) If $I(c, d) = \bot$ then $c \wedge d$ is unsatisfiable in A
   (b) If $I(c, d) = (c', \{d_1, \ldots, d_n\})$ then
      i. $A \models c \wedge d \Leftrightarrow c' \wedge d_1 \wedge \ldots \wedge d_n$
      ii. $A \models c' \Rightarrow c$

The solved constraints will play a prominent role in the calculus since they represent, in a given configuration, the knowledge about the values of the variables that can be freely used in the deductions. The role of the function I will be more clear with the following calculus.

3.2 A Refined CLP Calculus

Given a constraint system (L, C, A, S, I), a configuration is either FAIL or a pair $c \mid L$ where $c \in S$ (called the *constraint store*) and L is a list of atoms and constraints from C (the *agents*). The new calculus of constraint logic programming now comprises the rules depicted on Figure 4.

Some remarks on this calculus:

- The second component of the inference function may be empty. In this case the application of rule (Propagate) makes the constraint disappear from the collection of agents and integrates it completly in the constraint store.
- It follows from the properties of the inference function of the constraint system that the constraint store increases monotonously during computation: If $c \mid L \rightarrow^* c' \mid L'$ then we have that $A \models c' \Rightarrow c$.

$$\text{(Unfold)} \quad \frac{c \mid L, A(\bar{x})}{c \mid L, K} \qquad A(\bar{x}) \leftarrow K \text{ is a program clause.}$$

$$\text{(Fail)} \quad \frac{c \mid L, d}{\text{FAIL}} \qquad I(c, d) = \bot$$

$$\text{(Propagate)} \quad \frac{c \mid L, d}{c' \mid L, d_1, \ldots, d_n} \quad I(c, d) = (c', \{d_1, \ldots, d_n\})$$

Fig. 4. The rules of CLP with incomplete constraint solving.

- The conjunction of constraints "$\wedge$" is not to be confused with the aggregation operator "," of agents. A single agent $c \wedge c'$ does not necessarily behave the same way as two separate agents c and c' since the inference function can see only one agent at a time.

Given a set of *initial agents* L, the computation starts on `True` $\mid L$. A computation *succeeds* if it terminates in a configuration of the form $c \mid \emptyset$, in this case c is the *answer constraint*, it fails if it terminates in FAIL, and it is *stuck* if it terminates in a configuration where no rule applies (remind that the inference function is partial!).

3.3 Some Constraint Systems with Partial Solvers

Feature Constraints Our first example of a constraint system according to the new definition is CFT: The set of constraints and the structure are as in Section 2.2, and the the set of solved constraints is the set of constraints of the form $e \wedge c$ where c is a solved form and e is a set of equations that is eliminated in $e \wedge c$. The inference function is total in this case:

$$I_{CFT}(c, d) = \begin{cases} \bot & \text{if } c \wedge d \rightarrow^*_{CFT} \bot \\ c', \{\} & \text{if } c \wedge d \rightarrow^*_{CFT} c' \end{cases}$$

where $\rightarrow_{CFT}$ is the reduction relation introduced in Section 2.2. The fact that I_{CFT} is total and that, in case of non-failure, the second component is empty indicates that our constraint solver for CFT is complete.

The next example is the constraint system F that provided the motivation for incomplete constraint solving. We define the set of solved constraints as the set of constraints $e \wedge c$ where e is a set of equations eliminated in $e \wedge c$ (note that e may now give values to tree-variables *and* to feature-variables) and where c is a solved form of CFT (where we identify, slightly abusing notation, the F-constraint $x[f]y$ with the CFT-constraint $x[f]y$). The inference function is defined as follows:

$$I_F(c, d) = I_{CFT}(c, d) \qquad \text{if } d \text{ is a CFT constraint}$$

that is if d is one of $A(x)$, xF, $x[f]y$ with $f \in \mathcal{F}$ or an equation between tree-variables. The only other constraints that I_F can handle are equations for feature-variables:

$$I_F(c, v = t) = \begin{cases} \perp & \text{if } c[t/v] \text{ contains an equation between} \\ & \text{different feature symbols} \\ v = t \wedge c[t/v] & \text{otherwise} \end{cases}$$

$$I_F(c, x[v]y) = \begin{cases} (c, x[f]y) & \text{if } c \text{ contains an equation } v = f \\ \text{undefined} & \text{otherwise} \end{cases}$$

where v is a feature variable. Note that feature variables may occur in c only in the equational part, hence the condition is very easy to decide.

The crucial point here is that $I(c, x[v]y)$ is not defined if c contains no definition for v. In this case we say that the constraint $x[v]y$ is *suspended* since it is "waiting" for its feature variable to get defined by some other agent.

Nonlinear Equations Our last example is non-linear equations between integers. Constraints are conjunctions of one of the three forms $x = n$, $x = y + z$ and $x = y * z$ where n is an integer constant and x, y, z are variables. A conjunction containing only the first two forms of constraints is called *linear*. A constraint is solved if it is a linear system of the form

$$x_1 = t_1 \wedge \ldots \wedge x_n = t_n$$

where x_i is different from x_j for $i \neq j$, and x_i does not occur in $t_i, \ldots, t_n$. This is the well-known Gaussian triangular form. The equally well-known Gaussian elimination algorithm transforms any linear constraint c either into $\perp$ or into a solved form c', in this case we write $c \rightarrow^*_{lin} \perp$, resp. $c \rightarrow^*_{lin} c'$. Satisfiability of arbitrary constraints (involving products), however, is undecidable, as it has been shown by Matijacevič. We obtain an inference function for this constraint systems by suspending any non-linear equation $x = y * z$ until at least one of the two variables y and z is instantiated:

$$I_{NonLin}(c, d) = \begin{cases} \perp & \text{if } d \text{ linear, } c \wedge d \rightarrow^*_{lin} \perp \\ c', \{\} & \text{if } d \text{ linear, } c \wedge d \rightarrow^*_{lin} c' \end{cases}$$

$$I_{NonLin}(c, x = y * z) = \begin{cases} c, x = n * z \text{ if } c \text{ contains } y = n \\ c, x = n * y \text{ if } c \text{ contains } z = n \\ \text{undefined} \quad \text{otherwise} \end{cases}$$

The constraint $x = n * z$, for n an integer constant, can easily be expressed by linear constraints.

Is it possible to find a stronger inference function that can extract more linear constraints from a non-linear equation? The answer is yes, in fact there is an infinite growing chain of inference functions, since we can, for any $k = 1, 2, 3, \ldots$, add

$$I_{NonLin}(c, y = x * x) = c, \{y = x * x, y \geq 2kx - k^2\}$$

The inequation $y \geq 2kx - k^2$ can be seen as a linear constraint since it can be rewritten as

$$z = y + z_1 \wedge z_1 = k^2 \wedge z = 2kx + z'$$

where $z = 2kx + z'$ itself is a linear equation as we have seen above. The reader verifies easily the soundness of these inference functions.

Hence, the approach presented in this section leaves two open questions:

- What is a appropriate formalism to define the inference function of a constraint system?
- What inference function should we choose when there is no way to extract the "maximal" information from some agent, as in the case $y = x * x$ seen above?

In the next section we will present a programming formalism that allows the user, to some extent, to program the constraint system itself in the programming language. This also yields one possible answer to the second question since it gives the programmer the means to program constraint propagation (which corresponds to the inference function of this section) to be as strong as he needs it for his application.

To know more: The material of this section owes much to the survey [JM94]. See also [CDJK99] for a methodological view of constraint solving algorithms.

4 Committed Choice: A More Realistic Approach

The CLP calculus presented in Section 1.2 is an non-deterministic calculus. To define the semantics of of a CLP program it is necessary to talk about *all* possible deductions in the calculus. This non-determinism is not reflected in the calculus itself. For a concrete implementation of CLP this means that the machine must be able to explore systematically all computation paths, this is usually implemented with some kind of backtracking.

In this section we want to present another constraint-based programming formalism that

- represents indeterminism *explicitly* in the calculus, by using disjunctions, and
- allows both deterministic and non-deterministic computation steps.

4.1 A Calculus for Constraint Programming

Constraint Programming (CP) in its pure form follows a paradigm that is fundamentally different from the philosophy of CLP as seen in Section 1.2. CP is based on *deterministic* computation while the latter follows the tradition of *declarative programming*, that is based on *non-deterministic* computation that can be realized using an additional enumeration procedure.

In the following we give a simple calculus that reunites CLP and pure CP.

A *program* is a sequence of procedure definitions

$$P_1(x_1, \ldots, x_{n_1}) \Leftrightarrow A_1$$
$$\vdots$$
$$P_m(x_1, \ldots, x_{n_m}) \Leftrightarrow A_m$$

where the A_i are *agent expressions*. The variables x_i are considered *universally bound*. We will rename these bound variables whenever necessary. The set of agent expressions is defined by the following grammar:

$$\begin{array}{lll}
A & \rightarrow P(x_1, \ldots, x_n) & \textit{procedure call} \\
 & \rightarrow c & \textit{constraint} \\
 & \rightarrow A \wedge A & \textit{conjunction} \\
 & \rightarrow \exists \bar{x}\, A & \textit{scoped expression} \\
 & \rightarrow \texttt{if}\, D\, \texttt{fi} & \textit{conditional} \\
 & \rightarrow \texttt{or}\, D\, \texttt{ro} & \textit{disjunction} \\
D & \rightarrow & \textit{empty clause list} \\
 & \rightarrow C \mid D & \textit{non-empty clause list} \\
C & \rightarrow \exists \bar{x} c\, \texttt{then}\, A & \textit{clause}
\end{array}$$

In a clause list we usually omit the last $|$. In a clause $\exists \bar{x} c\, \texttt{then}\, A$, the scope of the existential quantifier extends over the constraint c (the *guard* of the clause) and the agent A (the *protected part* of the clause).

An important point of our version of the CP-calculus is that we adopt another solution than in Section 3 to cope with incomplete constraint solvers. Here, we assume that we dispose of a constraint system with a *complete* constraint solving algorithm, that is a constraint system in the sense of Section 1.3. Data descriptions that are not part of the constraint system in case we don't have an (efficient) constraint solver for them are now expressed by agent expressions called *constraint propagators*. Hence, what we treated in Section 3 as a constraint system with an incomplete inference function is now split up into one part which makes up the constraint system, and another part that is to be *programmed* (we will see soon how this can be accomplished).

A CP-configuration is either FAIL or a pair $c \mid As$ where c is a constraint and As is a set of agent expressions. The reduction rules of the calculus are given in Figure 5. In the Local-rule it is understood that $\bar{y}$ and $\bar{x}$ are both vectors of pairwise different variables and are of the same length. The notation $A[\bar{y}/\bar{x}]$ denotes the agent expression obtained by replacing in A every free occurrence of a free variable from $\bar{x}$ by its counterpart from $\bar{y}$.

A computation is called *successful* when it terminates in a configuration with an empty set of agents, it *fails* when it terminates in FAIL, and it is *stuck* when it terminates in a configuration where no further rule application is possible and where the agent space is non-empty.

There are three ways how a disjunctive agent can evolve:

1. A clause is dropped by the rule DisDrop if its guard is not consistent with the constraint store.

(Unfold)	$\dfrac{c \mid P(x_1,\ldots,x_n),\psi}{c \mid A,\psi}$	$P(x_1,\ldots,x_n) \Leftrightarrow A$ is a program clause
(Failure)	$\dfrac{c \mid c',\psi}{\textsc{Fail}}$	$c \wedge c'$ unsatisfiable
(Simplify)	$\dfrac{c \mid c',\psi}{c'' \mid \psi}$	$c \wedge c'$ satisfiable and $c \wedge c' \Leftrightarrow c''$
(Unpack)	$\dfrac{c \mid A_1 \wedge A_2,\psi}{c \mid A_1, A_2,\psi}$	
(Local)	$\dfrac{c \mid (\exists \bar{x}\, A),\psi}{c \mid A[\bar{y}/\bar{x}],\psi}$	$\bar{y}$ fresh variables
(DisDrop)	$\dfrac{c \mid \;\texttt{or}\; D \mid \exists \bar{x}\, c' \,\texttt{then}\, A \mid D' \,\texttt{ro}\,,\psi}{c \mid \;\texttt{or}\; D \mid D' \,\texttt{ro}\,,\psi}$	$c \wedge \exists \bar{x} c'$ unsatisfiable
(DisSingle)	$\dfrac{c \mid \;\texttt{or}\; \exists \bar{x}\, c \,\texttt{then}\, A \,\texttt{ro}\,,\psi}{c \mid (\exists \bar{x}\,(c,A)),\psi}$	
(DisEmpty)	$\dfrac{c \mid \;\texttt{or ro}}{\textsc{Fail}}$	
(CondDrop)	$\dfrac{c \mid \;\texttt{if}\; D \mid \exists \bar{x}\, c' \,\texttt{then}\, A \mid D' \,\texttt{fi}\,,\psi}{c \mid \;\texttt{if}\; D \mid D' \,\texttt{fi}\,,\psi}$	$c \wedge \exists \bar{x} c'$ unsatisfiable
(CondFire)	$\dfrac{c \mid \;\texttt{if}\; D \mid \exists \bar{x}\, c' \,\texttt{then}\, A \mid D' \,\texttt{fi}\,,\psi}{c \mid (\exists \bar{x}(c',A)),\psi}$	$c \models \exists \bar{c}'$,
(CondEmpty)	$\dfrac{c \mid \;\texttt{if fi}\,,\psi}{c \mid \psi}$	

Fig. 5. Computation rules for CP.

2. If the set of clauses is empty then the agent fails. This is consistent with the logical view of an empty disjunction being equivalent to $false$.
3. If there is only one clause in the agent then the disjunction disappears and sets free the guard together with its protected part. Application of the Local-rule will eliminate the existential quantifier, allowing for further consideration of the agent expressions.

This treatment of disjunction is very similar to the *Andorra* rule proposed by D. H. Warren for logic programming to allow determinate reduction steps (the names comes from And-Or).

The difference between the operational behavior of a disjunction and a conditional expression can be seen when comparing the rules DisSingle and CondFire: A conditional *fires* its clause if its guard is entailed by the constraint store. Entailment is necessary for firing the clause even when there is only one clause left in the conditional expression. The conditional vanishes when no clause is left; as an alternative to our definition, one could allow a conditional expression to have an *otherwise* expression which fires when all regular clauses are eliminated.

A dramatic consequence of the rules for the conditional expression is that, a priori, logical semantics is lost. For a given constraint store it might well be the case that the guards of several clauses are entailed at a time. In this case, our calculus makes an non-deterministic choice of the clause to be fired. This choice is fundamentally different from the choice seen in Section 1.2 when selecting a clause since now all non-determinism is explicit in the expressions. There are, however, ways to ensure by a careful analysis that a conditional agent still behaves in a logically meaningful way (see, for instance, [AFM96]).

The constraint store increases monotonically during the course of a computation, that is if $c \mid \psi \rightarrow^* c' \mid \psi'$ then $c \models c'$ holds. The consequence is that if in a certain configuration a conditional agent can fire one of its clauses, then he will still be able to fire this clause after further progress of the other agents (provided that the clause has not disappeared due to inconsistency). This is one key to consider the calculus as a calculus of *concurrent computations*.

The agents co-exist independently from each other in the agent space, the only way to have some "communication" among agents is by having them impose constraints on the constraint store. Formally, if $c \mid A, \psi \rightarrow c \mid \phi, \psi$, then for all sets ψ' of agent expressions we have as well that $c \mid A, \psi' \rightarrow c \mid \phi, \psi'$. This is another key to concurrency.

Let us now demonstrate the difference between disjunctions and conditionals with an example: the concatenation of lists. First we define a concatenation predicate as a disjunction (the binary constructor symbol ":" is here written in infix notation):

```
conc(X,Y,Z) ⇔
   or X=nil ∧ Y=Z
    |  ∃ H,X',Z' X=H:X' ∧ Z=H:Z' then conc(X',Y,Z')
   ro
```

Here is a query leading to a successful computation:

$$
\begin{array}{l}
X = a : nil \wedge Y = b : c : nil \mid conc(X, Y, Z) \\
X = a : nil \wedge Y = b : c : nil \mid \texttt{or}\ X = nil \wedge Y = Z \mid \exists H, X', Z' \ldots \texttt{ro} \\
X = a : nil \wedge Y = b : c : nil \mid \\
\qquad \texttt{or}\ \exists\, H, X', Z' X = H : X' \wedge Z = H : Z'\ \texttt{then}\ conc(X', Y, Z')\ \texttt{ro} \\
X = a : nil \wedge Y = b : c : nil \mid \exists H, X', Z'(X = H : X' \wedge Z = H : Z', conc(X', Y, Z')) \\
X = a : nil \wedge Y = b : c : nil \mid X = H : X' \wedge Z = H : Z', conc(X', Y, Z') \\
X = a : nil \wedge Y = b : c : nil \wedge H = a \wedge X' = nil \wedge Z = a : Z' \mid conc(X', Y, Z') \\
\ldots \\
X = a : nil \wedge Y = b : c : nil \wedge H = a \wedge X' = nil \wedge Z = a : b : c : nil \mid \emptyset
\end{array}
$$

The following computation, however, is not successful since it gets stuck with a disjunction that can not reduce any further:

$$
\begin{array}{l}
Y = nil \mid conc(X, Y, Z) \\
Y = nil \mid \texttt{or}\ X = nil \wedge Y = Z \mid \exists H, X', Z'\ X = H : X' \wedge \ldots \texttt{ro}
\end{array}
$$

This situation can always occur when we use disjunctive agents. One solution is to consider a *disjunction of configurations* and to have an additional enumeration rule in the calculus:

$$
\text{(Enumerate)}\ \frac{c \mid \texttt{or}\ C \mid D\ \texttt{ro}\ , \psi}{\Big(c \mid C, \psi\Big) \vee \Big(c \mid \texttt{or}\ D\ \texttt{ro}\ , \psi\Big)}
$$

This rule should only be used when no other reduction is possible. In some cases, in particular for constraint satisfaction problems as seen in Section 1.5, such an *enumeration rule* is necessary. In other cases, however, it is possible to overcome the weakness of disjunctions by using conditional expressions, as in our example. Here is a version of the concatenation predicate using a conditional agent expression:

```
conc(X,Y,Z) ⇔
  if X=nil then Y=Z
   | Y=nil then X=Z
   | Z=nil then X=nil ∧ Y=nil
   | ∃ H,X',Z' X=H:X'  then  Z=H:Z' ∧ conc(X',Y,Z')
  fi
```

With this conditional version of conc, both example queries seen above lead to successful computations:

$$
\begin{array}{l}
X = a : nil \wedge Y = b : c : nil \mid conc(X, Y, Z) \\
X = a : nil \wedge Y = b : c : nil \mid \\
\qquad \texttt{if}\ \ldots \mid \exists H, X', Z'\ X = H : X'\ \texttt{then}\ Z = H : Z' \wedge conc(X, Y, Z)\ \texttt{fi}
\end{array}
$$

$X = a : nil \wedge Y = b : c : nil \mid \exists H, X', Z'(X = H : X' \wedge Z = H : Z', conc(X', Y, Z'))$
$X = a : nil \wedge Y = b : c : nil \mid X = H : X' \wedge Z = H : Z', conc(X', Y, Z')$
$X = a : nil \wedge Y = b : c : nil \wedge Z = H : Z' \wedge H = a \wedge X' = nil \mid conc(X', Y, Z')$
$X = a : nil \wedge Y = b : c : nil \wedge Z = H : Z' \wedge H = a \wedge X' = nil \mid$
$\quad$ if $X' nil$ then $Y = Z$ fi
$X = a : nil \wedge Y = b : c : nil \wedge Z = H : Z' \wedge H = a \wedge X' = nil \wedge Z = b : c : nil \mid \emptyset$

$Y = nil \mid conc(X, Y, Z)$
$Y = nil \mid$ if $\ldots \mid Y = nil$ then $X = Z \mid \ldots$ fi
$Y = nil \mid Y = nil, X = Z$
$Y = nil \wedge X = Z \mid \emptyset$

4.2 Constraint Propagators for Boolean Constraints

Boolean constraints are an example where one constraint propagator makes use of an other constraint propagator. We give here the propagators for exclusive-or and inequality:

```
XOr(x,y,z) ⇔
if x=T then NEq (y,z) | y=T then NEq (x,z) | z=T then NEq(y,z)
 | x=F then y=z | y=F then x=z | z=F then x=y
 | x=y then z=F | x=z then y=F | y=z then x=F
fi
```

```
NEq(x,y) ⇔
if x=F then y=T | x=T then y=F | y=T then x=F | y=F then x=T
 | x=y then FAIL
fi
```

4.3 Constraint Propagators for Feature Constraints

The constraint propagator for the subtree constraint can be realized by testing a variable to be determined:

```
Subtree(X,V,Y) ⇔
  if Det(V) then X[V]Y
  fi
```

In case of the feature constraint system considered in this section, a conditional clause guarded by $Det(x)$ can fire if the x gets bound to some value. More generally, one defines that a constraint ψ determines a variable x if

$$\forall \bar{y} \exists^{\leq 1} x\ \psi \qquad \text{where } \bar{y} = V(\psi) - \{x\}$$

holds in the interpretation of the constraint system. In this formula, $\exists^{\leq 1} x\ \psi$ means *there is at most one value of x that satisfies ψ*, this formula can be seen as an abbreviation for

$$\forall x, x'(\psi \wedge \psi[x'/x] \Rightarrow x = x')$$

In case of the CFT constraint system this notion of a determined variable coincides exactly with the definition given in Section 2.2.

An important remark is that the expression $Det(x)$ is not a constraint since this expression does not have a meaning in itself, it rather is an observation *about the constraint store*. Strictly speaking, this is a violation of the formation rules of our calculus, nevertheless, such a special guard $Det(x)$ is easily integrated in the implementation of a constraint programming system.

More generally, we can see Det as a *reflection predicate* that can observe the state of the constraint store, thus raising in some sense the constraints to first-class status. Reflection of the constraint store raises the question of how to define proper semantics, however, it seems to be unavoidable if one wants to express useful constraint propagators in the programming calculus.

4.4 Finite Domain Constraints

We have already briefly encountered finite domain constraints in Section 1.5. We now give a complete definition.

An *interval* is a set of natural numbers of the form $\{n, n+1, \ldots, m\}$ where $m > n$. A *finite domain* is a finite union of intervals.

The language of finite domain constraints contains, for every finite domain D, a unary predicate $x \in D$ and an equality predicate $=$. Domains are written by using the notion $[n, m]$ for intervals and the symbol $\cup$ for union, as in $x \in [2,4] \cup [6,8]$, furthermore we will use a notation like $\{1, 17, 42\}$ for sparse sets. Constraints are (probably existentially quantified) conjunctions of atomic constraints, and the interpretation is the obvious one. An algorithm for checking satisfiability of constraints is straightforward (see Figure 6).

What is missing to make finite domain constraints useful is constraints that express some non-trivial relation between variables, as for instance $x \neq y, x = y + z, x = y * z, \ldots$. Unfortunately, deciding satisfiability of finite domain constraints together with addition $x = y+z$ is NP-complete even when only domains of the form $[0,1]$ are involved, as the reader may easily prove as an exercice (Hint: look at the proof of Theorem 7). Hence, we have another case where we need to program constraint propagators. The question is, however, how strong a constraint propagation we want to program. We will look at two approaches: domain propagation and interval propagation. Both propagation strategies are defined by reference to a logical reading. To keep things simple we will just look at one propagator $Plus(x, y, z)$, having a logical reading $x = y + z$.

The propagation methods that we present here always consider one agent expression $Plus(x, y, z)$ at a time, according to the principles which underly our programming model (see the remarks on the calculus above). In the literature on

$$(\text{Intersect})\ \frac{x \in D_1 \wedge x \in D_2 \wedge \psi}{x \in D_1 \cap D_2 \wedge \psi} \text{ if } D_1 \cap D_2 \neq \emptyset$$

$$(\text{Disjoint})\ \frac{x \in D_1 \wedge x \in D_2 \wedge \psi}{\perp} \text{ if } D_1 \cap D_2 = \emptyset$$

$$(\text{Eq})\ \frac{x = y \wedge \phi}{x = y \wedge \phi[y/x]} \quad x \text{ occurs in } \phi,\ x \neq y$$

$$(\text{Triv})\ \frac{x = x \wedge \phi}{\phi}$$

Fig. 6. Rules for Finite Domain Constraints

finite domain constraints this is called *local consistency* checking. In many application, however, it is necessary to check several agents simultaneously, that is to perform *global consistency* checking. This is often expressed by agent expressions acting on an arbitrary number of variables, for instance *AllDifferent*$(x_1, \ldots, x_n)$ which expresses, for arbitrary n, that all variables $x_1, \ldots, x_n$ take different values from their respective domains. This could of course be expressed as $n * (n-1)$ binary inequalities, however, treating all these inequalities at once in an *AllDifferent* expression is much more efficient.

Domain Propagation The definition of *domain propagation* is that (with the example of *Plus*):

$$\begin{aligned} & x \in D_x \wedge y \in D_y \wedge z \in D_z \ \mid\ Plus(x,y,z) \\ \rightarrow\ & x \in D'_x \wedge y \in D'_y \wedge z \in D'_z \ \mid\ Plus(x,y,z) \end{aligned}$$

such that the tuple (D'_x, D'_y, D'_z) is maximal (with respect to component-wise set inclusion) with

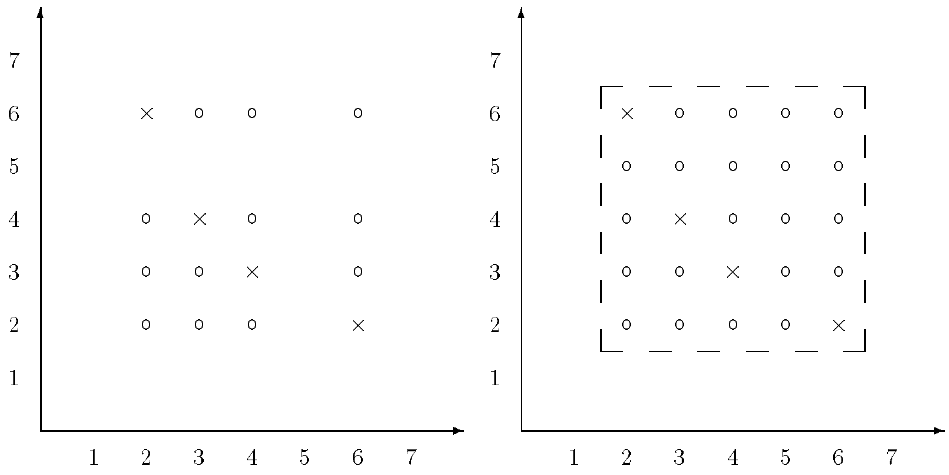
1. $D'_v \subseteq D_v$ for all variables v, and
2. for every $k \in D'_x$ there are $l \in D'_y$, $m \in D'_z$ such that $k = l + m$, and analogously for the other variables.

If this definition makes one of the domains empty then propagation of course yields the configuration Fail. For example,

$$\begin{aligned} & x \in [2,12] \wedge y \in [2,12] \wedge z \in \{12\} \ \mid\ Mult(x,y,z) \\ \rightarrow\ & x \in \{2,3,4,6\} \wedge y \in \{2,3,4,6\} \wedge z \in \{12\} \ \mid\ Mult(x,y,z) \end{aligned}$$

The situation is depicted in the left part of Figure 7. Note that in the diagram, every row and every column contains at least one $\times$-mark, due to the second condition of the above definition of domain propagation.

Furthermore it should be observed that with finite domain constraints (independent of the particular propagation method) the point-wise dependencies between variables are lost. This kind of abstraction is very similar to the set constraints considered in constraint-based program analysis (see Section 1.7).



(a) Domain Propagation

(b) Interval Propagation

Fig. 7. Constraint store found by different propagation methods. Only the values for x and y are shown. Pairs that are actual solutions are marked by $\times$.

Interval Propagation In comparison with domain propagation, *interval propagation* propagates less constraints (that is, leads in general to a weaker constraint store) but is computationally less expensive. For the sake of simplicity let us assume that all finite domain constraints in the store are interval constraints $x \in [n, m]$, it is obvious how to generalize to arbitrary finite domains.

$$\begin{aligned} & x \in [x_l, x_u] \wedge y \in [y_l, y_u] \wedge z \in [z_l, z_u] \mid Plus(x, y, z) \\ \rightarrow & x \in [x'_l, x'_u] \wedge y \in [y'_l, y'_u] \wedge z \in [z'_l, z'_u] \mid Plus(x, y, z) \end{aligned}$$

where for all variables v the values v'_l are maximal and the values v'_u are minimal such that

- for all variables v we have that $v_l \leq v'_l$ and $v'_u \leq v_u$, and
- There are $y_1, y_2 \in [y'_l, y'_u]$, $z_1, z_2 \in [z'_l, z'_u]$ such that $x'_l + y_1 = z_1$ and $x'_u + y_2 = z_2$, and analogously for the other variables.

If this definition makes one of the domains empty then propagation of course yields the configuration Fail. Alternatively, one could say that interval propagation yields the smallest intervals that subsume the sets found by domain propagation. We demonstrate this behavior with the same example that we used to demonstrate domain propagation:

$$\begin{aligned} & x \in [2,12] \wedge y \in [2,12] \wedge z \in \{12\} \mid Mult(x,y,z) \\ \rightarrow\ & x \in [2,6] \wedge y \in [2,6] \wedge z \in \{12\} \mid Mult(x,y,z) \end{aligned}$$

The situation is depicted in the right part of Figure 7. Note that the marked area is a rectangle, and that the rows and columns *along the border of the rectangle* each contain at least one $\times$-mark. Rows and columns in the interior of the rectangle, however, may not contain $\times$-marks.

4.5 Constraint Propagators for Interval Propagation

In order to express interval propagation in our calculus for constraint programming we need some way to know the actual bounds of a variable, that is to *reflect* the current state of the constraint store. For this purpose we will allow, for any variable x, in the guard of a conditional additional identifiers $x \uparrow$ and $x \downarrow$, denoting the current lower and upper bound of the domain of x. This is very similar to the predicate *Det* that we employed in Section 4.3, note that an expression using the notation $x \uparrow$ or $x \downarrow$ is not a constraint in the proper sense since it makes a statement about the current constraint store itself instead of about the value of the variable x.

Here is a possible definition for the propagator *Plus*:

```
Plus(x,y,z) ⇔
if x↑ + y↑ < z↑ then z ∈ [z↓,x↑+y↑] ∧ Plus(x,y,z)
 | x↓ + y↓ > z↓ then z ∈ [x↓+y↓,z↑] ∧ Plus(x,y,z)
 | x↓ + y↑ < z↓ then x ∈ [z↓-y↑,x↑] ∧ Plus(x,y,z)
 | x↑ + y↓ > z↑ then x ∈ [x↓,z↑-y↓] ∧ Plus(x,y,z)
 | x↑ + y↓ < z↓ then y ∈ [z↓-x↑,y↑] ∧ Plus(x,y,z)
 | x↓ + y↑ > z↑ then y ∈ [y↓,z↑-x↓] ∧ Plus(x,y,z)
 | x↓ = x↑ ∧ y↓ = y↑ then z = x↓ + y↓
 | x↓ = x↑ ∧ z↓ = z↑ then y = z↓ - x↑
 | y↓ = y↑ ∧ z↓ = z↑ then x = z↓ - y↓
fi
```

4.6 Concluding Remarks

In this section we have tried to demonstrate how constraint propagation can be expressed in a calculus based on constraint programming. However, there is no final answer to the question how to integrate constraints and programming. We have seen that already for quite simple constraint systems such as feature constraints with features as first-class values, or finite domain constraints with arithmetical relations between variables, we needed reflection of the constraint store as an extension of the programming calculus.

Programming constraint propagation in the programming calculus itself is not the only way to realize constraint propagators, constraint propagation might well be expressed in a dedicated formalism. Efficiency consideration will often lead to hardwire constraint propagation into a programming environment. Nevertheless, a programmer needs to know exactly the constraint propagation mechanism; a high-level description in the programming calculus might hence be a good way to specify the mechanism.

Last but not least, it is not obvious how constraint-based programming integrates with other programming paradigms as functional programming, object-oriented programming, or how constraint-based calculations can be distributed over different machines. Two lectures of this summer school, *Functional and Constraint Logic Programming* and *Concurrent Constraint Programming based on Functional Programming*, will address programming calculi based on constraints.

To know more: Constraint calculi with committed choice go back to Saraswat's family of cc-languages, see for instance [SR91]. The material of this section reuses many ideas of the Oz project, the actual Oz-calculus, however, is much more expressive than the primitive calculus presented here, see [Smo95] and `http://www.mozart-oz.org`.

A good introduction to finite domain constraints is [Hen89]. More about real-life solutions by finite domain constraints will be told in the lecture on *Building Industrial Applications with Constraint Programming*.

References

[AFM96] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. On confluence of constraint handling rules. In Eugene Freuder, editor, *Principles and Practice of Constraint Programming*, volume 1118 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996.

[Apt90] Krzysztof R. Apt. Logic programming. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B - Formal Models and Semantics, chapter 10, pages 493–574. Elsevier Science Publishers and The MIT Press, 1990.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[CDG+97] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata. Techniques and applications, 1997. Preprint available at `http://www.grappa.univ-lille3.fr/tata/`.

[CDJK99] Hubert Comon, Mehmet Dincbas, Jean-Pierre Jouannaud, and Claude Kirchner. A methodological view of constraint solving. *Constraints*, 1999. To appear.

[Coh90] Jacques Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[Com95] Hubert Comon. On unification of terms with integer exponents. *Mathematical Systems Theory*, 28(1):67–88, January/February 1995.

[CT94] Hubert Comon and Ralf Treinen. Ordering constraints on trees. In Sophie Tison, editor, *Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science, vol. 787, pages 1–14, Edinburgh, Scotland, 11-13 April 1994. Springer-Verlag.

[DSH90] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:75–93, 1990.

[Hen89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[HJ90] Nevin Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 197–209, San Francisco, CA, January 1990.

[JK91] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*, chapter 8, pages 257–321. MIT Press, Cambridge, Massachusetts, 1991. Essays in Honor of Alan Robinson.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[Kel93] Bill Keller. *Feature Logics, Infinitary Descriptions and Grammar*. CSLI Lecture Notes No. 44. Center for the Study of Language and Information, 1993.

[Nie99] Robert Nieuwenhuis. Rewrite-based deduction and symbolic constraints. In Harald Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 302–313, Trento, Italy, July 1999. Springer-Verlag.

[PP97] Leszek Pacholski and Andreas Podelski. Set constraints: A pearl in research on constraints. In Gert Smolka, editor, *Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 549–561, Linz, Austria, October/November 1997. Springer-Verlag.

[Smo92] Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Current Trends in Computer Science*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, Heidelberg, New York, 1995.

[SR91] Vijay Saraswat and Martin Rinard. Semantic foundations of concurrent constraint programming. In *Proceedings of the 18th Symposium on Principles of Programming Languages*, pages 333–351, Orlando, FL, January 1991. ACM.

[ST94] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[Tre93] Ralf Treinen. Feature constraints with first-class features. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, vol. 711, pages 734–743. Springer-Verlag, 30 August–3 September 1993.