From Unification to Constraints

Joxan Jaffar and Jean-Louis Lassez
I.B.M. Thomas J. Watson Research Center

Abstract
The constraint paradigm plays a more and more important role in knowledge based systems and declarative programming. This is because because it caters for implicit information and the representation of fundamental domains of computation. In particular constraint solving can advantageously replace unification as it corresponds better to programming practice and existing trends in language design. Furthermore the introduction of constraints in Logic Programming preserves and enhances very naturally the desirable semantic properties of Logic Programs. We give here a brief exposition of the motivations that led to the CLP theory, an overview of the language CLP($\mathcal{R}$), an example of application in Stock Options Trading and finally we mention a number of important activities in the area of Constraints and Logic Programming.

Introduction

The Constraint Logic Programming (CLP) Scheme is a framework for the formal foundations of a class of programming languages. This framework is designed to encapsulate both the paradigms of constraint solving and logic programming, it is presented in Jaffar and Lassez [7, 8]. The constraint solving paradigm allows concise and natural representation of complex problems because of two main reasons:

(a) the constraints state properties directly in the domain of discourse as opposed to having these properties coded into Prolog terms or Lisp lists.
(b) the constraints have the ability of representing properties implicitly as opposed to having bindings to variables. The logic programming paradigm, on the other hand, suggests an overall rule-based framework to reason about constraints.

This article is intended as a brief exposition on three aspects of Constraint Logic Programming: for the theory aspect, we will discuss what it means for CLP to be a Scheme and how, consequently, all the resulting languages have the same essential semantic properties. It is important to note here that these semantic properties are the same as those of conventional Logic Programs. So the introduction of constraints in Logic Programming increases the expressive power

without a loss in semantic properties. For the implementation and application aspects, we will choose the particular domain of real arithmetic, and describe CLP($\mathcal{R}$), an instance of the scheme initially defined in Jaffar and Michaylov [9]. A more involved paper by Jaffar, Michaylov, Stuckey and Yap [10] is forthcoming. We highlight the above points that the CLP($\mathcal{R}$) arithmetic constraints are meant to be interpreted exactly in the usual way people understand arithmetic, and also that the ability to state arithmetic properties implicitly is indeed part of a powerful programming methodology.

We illustrate the expressive power of CLP($\mathcal{R}$) with an example in economics, taken from work by C. Lassez, K. McAloon and R. Yap [13]. In a last section, we indicate a number of recent works relating Constraints and Logic Programming and suggest or mention a number of potentially significant directions where substantial work is in progress.

We begin with an informal discussion on constraints in the context of programming.

Implicit and explicit information

Constraint solving is a paradigm that is widely used in graphics, engineering, knowledge representation, etc. as it allows great expressive power from a declarative point of view. Typically, we want to design systems over well understood domains such as sets, graphs, Boolean expressions, integers, rationals, real numbers, etc. These domains have natural algebraic operations associated with them such as set or graph intersection, disjunction, multiplication, etc. They also have associated privileged predicates such as set equality, graph isomorphism and various forms of inequalities such as set inclusion, $<$, $\geq$, $\neq$, etc. These predicates are examples of what we call constraints. Note that the various forms of equality are just a very particular type of constraints.

The key property of constraints is that they allow the user to define objects *implicitly*. Consider the following elementary examples:

(1) A set of names of people is given explicitly as a list $S = (John\ Doe, Barbara\ Smith, ...)$, this same set can also be defined implicitly with the constraints $S = \{x : salary(x) \geq 35000\ \&\ status(x) \neq manager\}$. The first definition is suitable for some operations like mailing letters to these people, the second definition, however, conveys information that is not apparent in the first one and would be more suitable if one wanted to design an automatic system to help in deciding increases in salaries.

(2) In a different domain, consider the implicit and explicit definitions of a typical conic: $ax^2 + by^2 + cxy + dx + ey + f=0$ and $x=u(t)/w(t), y=v(t)/w(t)$ where u, v, w are quadratic polynomials. The explicit representation is clearly the one to use to generate a picture of this conic, but to determine whether a given point is above or below the surface or curve the implicit representation is more appropriate.

Implicit and explicit representations play important and complementary role and to be able to pass from one representation to the other is a crucial problem in many fields. In particular, implicit representations are necessary when finite explicit representations are impossible to obtain. For instance, the set of points (x,y) defined by $x \neq y$ or $x \geq y$ cannot be represented explicitly in a finite way and a listing of these points is obviously out of the question, it would not even be countable. Let us mention why this point illustrates a fundamental weakness in conventional Logic Programs. Output from Logic Programs are substitutions, that is strictly of the explicit type, $x = f(u,v)$, $y = g(u)$ for instance. Negation on the other hand allows you to define sets implicitly, for instance $x \neq f(u,u)$. Such a set cannot be finitely represented by substitutions. Implications of this remark on explicit versus implicit representations have been discussed in Lassez and Marriott [14], Lassez, Maher and Marriott [15]. It is shown there that only in very restricted cases is it possible to transform an implicit representation (via terms and complements or system of equations and inequations) into a finite explicit one. It was also mentioned that a major shortcoming of the negation as failure rule comes from the fact that negative information appears implicitly in a program, but bindings are performed only on explicitly defined objects. So the problem arises of transforming implicit information into explicit form. But it is in general impossible to finitely replace the implicit information by an explicit one.

A conflict arises: we are allowed to state implicit information but can compute, input and output only in an explicit manner via unification. So the expressive power of the language we use to write programs allows us to reason about objects that we cannot output. In fact it is clear that in general adding negation to a language strictly adds to its expressive power, a further example is that the complement of a recursively enumerable set cannot be described as a recursively enumerable set. But even in cases where negation could be eliminated in principle it may be more natural or efficient to keep it. Consider the finite set {a,b,c,d,e,f,....,z} One could systematically replace expressions such as $x \neq a$ by $x = b \vee x = c \vee \ldots\ldots \vee x = z$. But the implicit definition using $\neq$ is certainly more economical and may convey a more useful type of information.

So the points we made here are that we may need or prefer for many applications to use implicit representations rather than restrict ourselves to explicit ones because it is more natural or because we in fact have no choice.

The CLP Scheme

If we introduce via constraints such implicit representations in Logic Programming, we can expect to derive languages with strictly more expressive power. A clue on how to do it is to replace the Herbrand universe by a user's domain of computation and unification by constraint solving. This is a fairly drastic change and it is not quite clear a priori to see how the formal semantics of logic programs will be affected. In fact the key semantic properties of Logic Programs are preserved in this generalization. We are making here a very strong statement namely that Logic Programming does not depend in any essential way on unification and the Herbrand universe, and that we can as well do Logic Programming on real numbers and arithmetic constraints, or sets or graphs and their associated constraints, to cite a few possibilities. To fully understand this, one of course has to go through all the theorems and proofs to be found in Jaffar and Lassez [8]. Nevertheless a good understanding of the difference between unification and constraint solving can provide an informal and clear clue to why the semantics are indeed preserved, as we show now.

There are two aspects to unification. One is that the unification algorithm is a decision procedure which tells us if the equation t1=t2 has a solution. The other aspect is that it gives us as output a substitution (mgu) which represents the set of all solutions. But one could argue that the equation t1=t2 also represents the set of all solutions. Why is the mgu needed? It is needed only for syntactic convenience, it is simpler to handle x=a rather than f(x,x)=f(a,a). But we could in principle refuse this syntactic simplification without affecting any of the soundness and completeness, fixed points and least model theorems or their proofs. The price to pay would be only that as output we would have a non-simplified system of equations instead of a more concise mgu. But of course they would be equivalent, representing the same solutions. The mgu is a finite explicit representation of the set of solutions, t1=t2 is an implicit representation of the same set. When we deal with a domain where an mgu (or finite collection of mgu's) exists then it is convenient to use, but we know that in many instances such a finite explicit representation does not exist. So what is fundamental here is the decision aspect of unification, not its mgu aspect. Which means the essential aspect of unification is its constraint solvability aspect. The same algorithm which tells us if t1=t2 is solvable also tells us if $t1 \neq t2$ is solvable. So $t1 \neq t2$ is

a constraint of the same standing as t1=t2, and both of them can appear in the input, program, goals, and output. Those familiar with the various theorems mentioned will realize that this use of the constraint $\neq$ effectively doe not essentially affect the theorems or their proofs. What essentially matters is that the constraint is solvable, the nature of the constraint and its representation are irrelevant. Similarly one could take other specific domains and constraints and check case by case, theorem by theorem. This remark and example should provide an intuitive justification why unification is just a particular case of constraint solving and that logic programming systems can use in the same way other types of constraints. Nevertheless a fair amount of work was needed to find an appropriate formalism to handle the general case.

The key concept of a scheme, first introduced in Jaffar, Lassez and Maher [6] plays a crucial role here and this is what we discuss now. An analogy with the notion of vector space will be useful to understand the notion of a scheme which provides in an abstract way the same semantics to a class of languages. We first have axioms which define the abstract notion of vector space. Any mathematical object which satisfies these axioms is a vector space. And we have a whole collection of theorems established in the abstract setting, that hold for all these objects. The power of this notion comes from the fact that objects of apparently very different nature such as R^n, sets of matrices, sets of polynomials, etc, are vector spaces and are shown to have substantial structural properties in common. So when we study a new mathematical object if we can verify first that it satisfies the axioms of vector space, we have for free a lot of theorems, which is more satisfactory than the brute force approach of establishing them one by one.

In our context of semantics what plays the role of the vector space axioms, is the abstract notion of *solution compact*. The collection of theorems include soundness and completeness results for success, finite failure and negation as failure, and various least and greatest fixpoint and model theorems. These results are used to formalize the declarative aspects, the operational aspects and establish their relationships. Therefore they provide a high level rigorous definition of a class of languages and their implementation.

Informally we consider a domain whose objects are described by a language of constraints. Essentially this domain will be solution compact if the language of constraints allows us to define limit elements and is precise enough that we can distinguish any object which does not satisfy given constraints from those objects which do. There are a number of interesting aspects to this notion. First it represents a very weak requirement, so that we can claim that if a domain and associated language of constraints was not solution compact, then we would not

consider it for computational purposes as the syntax would not allow a precise definition of the objects. All finite or countable domains are solution compact in a trivial manner. Uncountable domains such as reals and infinite trees are also solution compact. So if the notion of solution compact helps us establish the theorems of the scheme, it is important to note that the reverse is true, that is if the theorems of the scheme hold for a particular domain, then it has to be solution compact.

So we have a powerful tool to generate semantics for a variety of languages which cater for different domains but use the rule based paradigm of Logic Programming. We can choose these domains to be user's domains rather than artificial symbolic ones, and therefore enhance the important semantic properties of Logic Programs.

The CLP($\mathscr{R}$) Language and Operational Model

We will now describe a particular instance of the scheme, which caters for real arithmetic constraints. It was chosen as a first illustration of the scheme as the domain of real numbers is of great importance and, as far as we know there is no formally defined or implemented programming language that effectively deals with real numbers. The terms are built using uninterpreted functors and real arithmetic terms in such a way that the functors may have real arithmetic terms as arguments, but not vice versa. Constraints are built from the usual equality and inequality relations between real number terms, and equality between terms which may contain functors.

A CLP($\mathscr{R}$) program consists of a finite number of rules, and these are of the form:

$$A_0 \text{ :- } c_1 , c_2, \ldots, c_n, A_1 , A_2, \ldots, A_m.$$

where c_i, $0 \leq i \leq n$, are primitive constraints and A_i , $0 \leq i \leq m$, are atoms.

Thus rules in CLP($\mathscr{R}$) have much the same format as those in PROLOG except that an un-ordered collection of primitive constraints may appear together with atoms in the body. The same applies to a CLP($\mathscr{R}$) *goal*; this is of the form

$$?\text{- } c_1, c_2, \ldots , c_n, A_1 , A_2, \ldots , A_m.$$

where c_i, $0 \leq i \leq n$, are primitive constraints, A_i , $0 \leq i \leq m$, are atoms, and $n + m \geq 1$.

Let P denote a CLP($\mathscr{R}$) program. Let G_l be a goal containing the collection C_l of primitive constraints. We say that there is a *derivation step* from G_l to goal G_2 if G_l is of the form

$$?- C_1, A_1, A_2, \ldots, A_m.$$

where $m \geq 1$, C_l is solvable, and P contains a rule of the form

$$B_0 :- C_2, D_1 , D_2, \ldots , D_k.$$

where $k \geq 0$, and G_2 is of the form

$$?- C_1, C_2, A_i = B_0, A_1 , \ldots , A_{i-1}, D_1 , \ldots , D_k, A_{i+1} , \ldots , A_m.$$

where $1 \leq i \leq m$, and C_l , C_2, $A_i = B_0$, is solvable. We say that the A_i in G_l above is the *selected atom.* Equivalently, A_i is the subgoal of G_l chosen to be *reduced.*

A *derivation sequence* (or simply sequence) is a possibly infinite sequence of goals wherein there is a derivation step to each goal from the preceding goal. A derivation sequence is *successful* if it is finite and its last goal contains only constraints. Such constraints are called *answer constraints.* They constitute the output of CLP($\mathscr{R}$) programs, and facilitate an outstanding feature of the language: *symbolic* output. Finally, a *finitely failed* sequence is a finite sequence where no derivation step, using the selected atom, is possible from the last goal in the sequence.

Thus a CLP($\mathscr{R}$) program and goal are executed by a process of continuously reducing any remaining atoms in the subgoal. There are two nondeterministic aspects in obtaining a derivation sequence as we have defined above. An *atom selection rule* determines in a goal which atom should be reduced next. A *search strategy*, on the other hand, determines which rule is to be used in the reduction of a given atom in the subgoal. In present implementations CLP($\mathscr{R}$) uses the atom selection and search strategies as in PROLOG. Of course, other strategies may be considered without interfering with the theoretical aspects.

Consider an example of a CLP($\mathscr{R}$) program. This program relates some simple parameters in a mortgage. The feature displayed here is that CLP($\mathscr{R}$), via its answer constraints, can produce symbolic answers.

```
mortgage(P, Time, I, MP, B) :-
        0 < Time, Time ≤ 1,
        Int = Time * (P * I/1200),
        B = P + Int - (Time * MP).
mortgage(P, Time, I, MP, B) :-
        Time > 1,
        Int = P * I/1200,
        mortgage(P + Int - MP, Time - 1, I, MP, B).
```

The parameters considered above are the principal, life of mortgage (in months), interest rate (in %), monthly payment, and finally the outstanding balance. The amount of interest to be paid is adjusted monthly.

The following goal represents a typical query for the monthly payment given all the other parameters:

```
?- mortgage(123456, 120, 12, MP, 0).
```

The answer constraint MP = 1771.23 is obtained, as it can be from a straightforward rewrite of the above program in any other imperative language. CLP($\mathscr{R}$) however enjoys the advantage of being a relational language, and so we may ask the reverse question:

```
?- mortgage(P, 120, 12, 1771.23, 0).
```

and obtain P = 123456. These examples, however, fall far short illustrating the expressive power of CLP($\mathscr{R}$). Consider the following goal in which only the time and interest are given:

```
?- mortgage(P, 120, 12, MP, B).
```

The CLP($\mathscr{R}$) system will return the *relationship* between the principal, monthly payment and balance:

```
B = 0.302995*P - 69.700522*MP.
```

This answer may be viewed as another program obtained as a result of partially evaluating the original program with respect to T = 120 and I = 12.

For more examples, see [4].

We conclude this sub-section by emphasizing that constraints in the domain of $\mathscr{R}$ provide a natural implicit representation for real numbers. It is important to distinguish "real" from "rational" here. For example, one could argue that since we finitely represent numbers in computers, we should base our theory on rational numbers. It then would follow that the program and goal below

```
ohmlaw(V, I, R) :- V = I * R.
?-ohmlaw(2, X, X)
```

would lead to failure (because $X^2=2$ has no rational solution) when it clearly should succeed. More generally, with constraints we can represent exactly, and reason about real numbers such as square root of two, defined as $x^2=2, x \geq 0$.

The CLP($\mathscr{R}$) Interpreter

In principle, one can build a CLP system from an easy marriage between a PROLOG-like engine and a constraint solver for the kinds of constraints at hand. In practice, this is not feasible. This is primarily because general purpose solvers are typically aimed at large and static problems. In CLP($\mathscr{R}$), on the other hand, we are dealing with dynamically created and typically smaller and specialized kinds of constraints. The number of such constraints, however, may be very large.

As indicated by the examples above, two central observations dominate the implementation issues in building any CLP system:

- each derivation step requires a solvability test for a collection of constraints, and
- these collections progressively grow larger.

It follows that two properties below are important to have in the handling of constraints:

- ***incrementality:***
  This means that whenever a collection of constraints is determined to be solvable, the solvability problem resulting from adding a new constraint does

not necessitate the repetition of all the work already done in determining the solvability of the original collection.

- ***canonicality:***
  This is taken to mean that there is standard, simplified representation of the collection which uses a minimal number of constraints. Appropriate canonical representations can have numerous advantages, for example, they can enhance efficiency because all equalities entailed by the collection will in fact appear explicitly in the collection.

It should be pointed out here that one can take for granted that equation solving algorithms are canonical and return a simple and useful canonical form. However for more general constraints the problems of finding incremental algorithms and appropriate canonical forms maybe very complex, this will be discussed in forthcoming publications. The underlying philosophy behind the CLP($\mathscr{R}$) implementation is that a priority is given upon several classes of constraints, and this priority roughly reflects both the computational cost and the expected relative frequency of constraints in a given class. For example, an equation between two variables is of the highest priority whilst a multivariate polynomial equation is of the lowest.

The first four of the five modules below (see figure 1) reflect our four priority classes:

- the ***engine***;
  This module is at the center of the entire system and is based upon a standard PROLOG engine. The important feature here concerning constraints is that the engine deals with the highest priority constraints without invoking any other modules.
- the ***interface***;
  This module transforms constraints into a standard form. This is done mainly for software engineering reasons, to help make the engine and the constraint solver relatively independent of each other. Like the engine, the interface does some constraint satisfaction on its own, thus lessening the need to use the solvers.
- the ***linear equality solver***;
  This module keeps a satisfiable collection of linear equations in parametric solved form, that is, some variables arising from program execution are described in terms of parametric variables. The method of solution used is a variant of Gaussian elimination. This module controls, and contains the only calls to, the linear inequality solver.

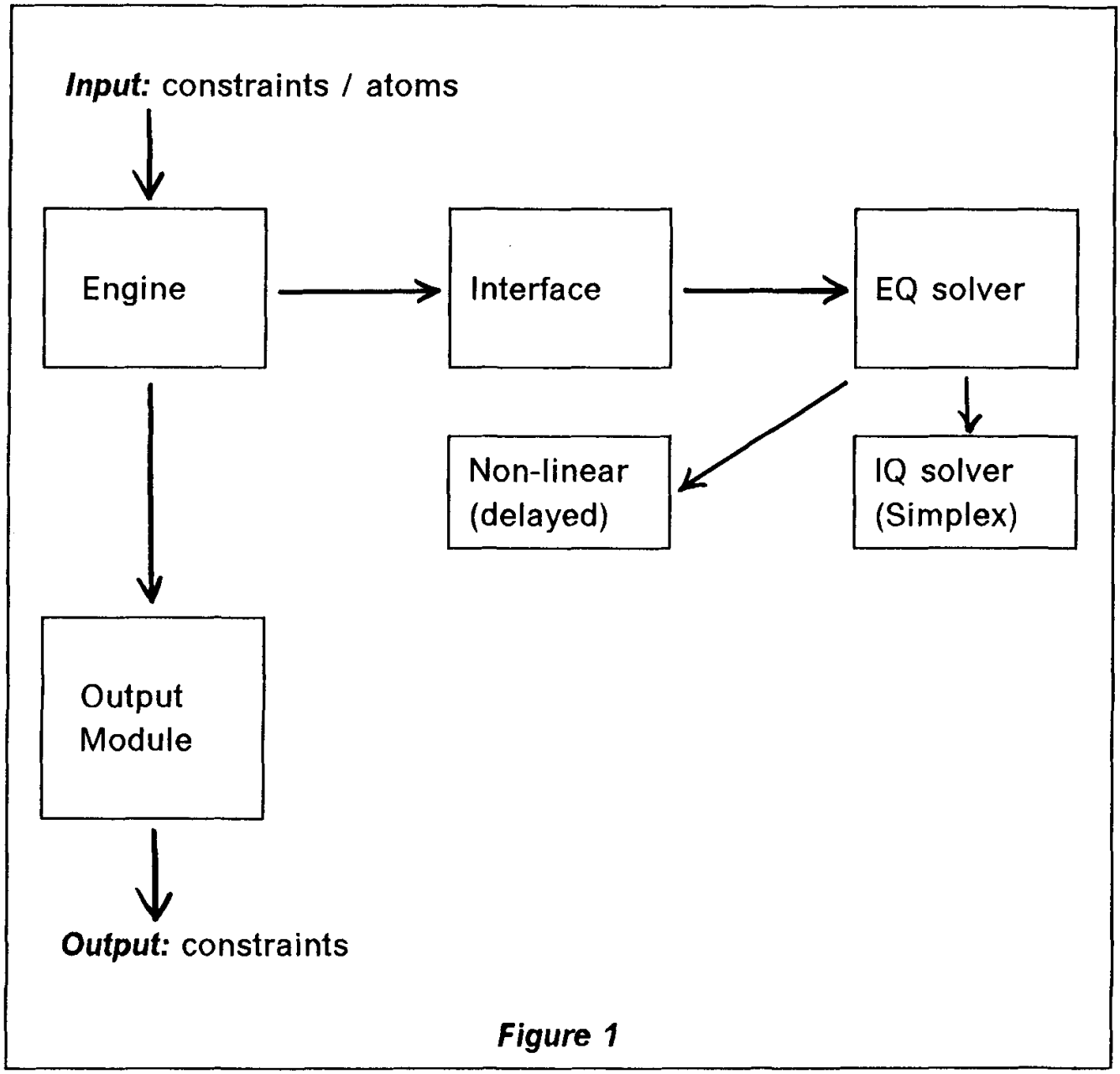

Figure 1

- the ***linear inequality solver***;
  This module is implemented using a sparse-matrix representation of a Simplex tableau and a modified form of the Simplex algorithm. It may operate without invoking any other module. However, it can, for efficiency, communicate a certain amount of information back to the equality solver. It does this whenever an equality is inferred from the collection of (non-strict) inequalities.
- the ***output*** module;
  The purpose of this module is two-fold. Firstly it de-parameterizes the constraints as much as possible so that answer constraints reflect relationships only between the variables in the goal. It then transforms these resulting constraints into a canonical form. This serves both the purpose of standard-

ization, e.g. two sets of constraints defining the same solution space should look alike, and the purpose of having the output as compact as possible.

Non-linear equations, in the present implementations, are not checked for solvability. Instead, they are stored away and inspected only when a sufficient number of variables in a non-linear equation become ground so that it becomes linear. This "delaying" and "waking" activity is controlled by the equality solver.

An example: option pricing

The world of finance is a great user of computer power. As many financial applications contain a mixture of mathematical computation and expert knowledge, they are well suited to a language like CLP($\mathscr{R}$). We present here some elements of a program that determines the price of stock options and which illustrates well the unique nature of the language. The following is derived from [12].

An option gives its owner the right to buy or sell a particular stock at a given price for a limited time. For instance an XYZ July 50 call option is the right to buy 100 shares of XYZ stock at $50 per share until the July expiration date regardless of the market price of the XYZ stock. Similarly an XYZ July 45 put option is the right to sell 100 shares of XYZ stock at $45 per share. The *Black-Scholes* formula (9) given below computes the theoretical price of an option in terms of stock price (p), strike price (the exercise price of the option) (s), time to expiration (t), current risk-free interest rate (r) and volatility (v) of the underlying stock measured by annual standard deviation:

$$\text{Theoretical option price} = pN(d_1) - se^{-rt}N(d_2)$$

$$d_1 = \frac{ln(\frac{p}{s}) + (r + \frac{v^2}{2})t}{v}\sqrt{t}\ ,\ d_2 = d_1 - v\sqrt{t}$$

(*ln* is the natural log and $N(x)$ the cumulative normal distribution).

The Black-Scholes formula is written as a CLP($\mathscr{R}$) clause

```
black_scholes(Th,P,S,T,R,V) :-
      Th = P * normal(D1) - S * exp(-R*T) * normal(D2),
      D1 = (ln(P/S) + (R + pow(V,2)/2 * T)/(V * sqrt(T)),
      D2 = D1 - V * sqrt(T).
```

where normal, exp, ln, pow and sqrt are system functions. The next two goals return respectively the option price and the stock price for a given option.

```
?-black_scholes(X,40,44,90,0.15,0.60).

---------------------
answer ---> X = 8.129


?-black_scholes(8.129,40,X,90,0.15,0.60).

------------------
answer ---> X = 44
```

This simple program can be incorporated in a larger one implementing for instance a buying strategy that typically requires the computation of the expected returns on all options in a given class. CLP($\mathcal{R}$) inherits from Logic Programming its incrementality which makes the task of expanding an existing program easy. In the Black-Scholes model, one parameter of particular importance is the volatility which is not always known. It can be computed statistically but the necessary data on the underlying stock are not always available and furthermore studies have shown that the statistical model is not always accurate. However, the volatility can also be computed using the Black-Scholes model under what is known as the *fair market assumption* which assumes that the actual option price equals the theoretical price. The Black-Scholes formula can then be used *in reverse* to compute the volatility. This requires solving a non-linear equation, but this is easily programmed in CLP($\mathcal{R}$) using a standard method to find the roots of a non-linear equation. This is illustrated in the next example which uses Steffensen's formula:

$$x_{n+1}=x_n-\frac{f(x_n)}{g(x_n)},\ g(x_n)=\frac{f(x_n+f(x_n))-f(x_n)}{f(x_n)}$$

The clause

```
getnext(X, NX) :-
     NX = X - F/G
     eval(X, F),
     eval(X + F, FF),
     G = (FF - F)/F.
```

expresses the formula with X = x_n, NX = x_{n+1}. The predicate eval is defined by:

```
eval(X, F) :-
     black_scholes(Th,P,S,T,R,X),
     market_price(Ma),
     F = Ma - Th.
```

The Black-Scholes formula is replaced by the equation

$$F=Ma - pN(d_1) - se^{-rt}N(d_2)=0$$

which is solved for the volatility. This is done by the routine solve which takes an initial value for X and an accuracy factor Epsilon.

```
solve(X, Epsilon) :-
     eval(X, F),
     abs(F) < Epsilon.
solve(X, Epsilon) :-
     getnext(X,NX),
     solve(NX, Epsilon).
```

It terminates successfully when the value of F for the last computed value of X is less than the accuracy factor Epsilon otherwise it calls itself recursively with NX, the next value of X.

Finally, a new clause is added to the previous definition of black_scholes to deal with the case when the volatility is unknown. Provided the market price of the option is given as well as the parameters for solve, the system will compute the volatility as is shown next (unknown succeeds when its parameter is ground -has been given a value- and fails otherwise).

```
black_scholes(Th,P,S,T,R,V) :-
      unknown(V), initial_value(X0),
      accuracy(E), solve(X0,E).


?-black_scholes(_,40,X,90,0.15,V), market_price(8),
  initial_value(0.10), accuracy(0.0005).


--------------------
answer ---> V = 0.598
```

An options buying program must combine expertise to develop strategies with numerical computation to evaluate parameters of those strategies. CLP($\mathcal{R}$) which combines the expressive power of logic programs with the numeric computational capability of a traditional algorithmic language is particularly well suited for this type of applications. A more thorough treatment for this application is to be found in [13].

Other domains of applications have been investigated and interesting applications have been reported in the domain of electrical engineering [3]. These early reports confirm the real usability of CLP($\mathcal{R}$) in domains that were up to now considered to be outside the range of logic programming languages.

Future Directions

Constraint solving is a topic of great importance in Artificial Intelligence, Mathematical Programming and Operations Research. It is clear that progress in these fields will have a deep influence on the CLP theory, on its algorithmic aspects and implementations. Preliminary results will be found in forthcoming papers Lassez and McAloon [16] and Lassez, Maher and Stuckey [17]. At that point in time we do not want to elaborate on these connections as the problem is far too involved, but it opens major new research directions. Our aim here will be more limited and we will suggest or report on recent developments which have a direct connection with CLP.

First as we previously mentioned, many extensions to Prolog can be viewed as instances of the CLP scheme, even though they were not necessarily conceived in that framework. A most notable one is Colmerauer's Prolog-III, which caters for rational arithmetic constraints and boolean constraints [1]. Another language is Mukai's CIL [22]; it is of great interest as is caters for a new and challenging type of constraints of non-numerical nature. Work is in progress to formally establish the connection with the CLP scheme. The work done by Van Hentenryck and Dincbas at ECRC is of particular interest (see e.g. [24]) as they attack successfully real industrial problems, such as cutting stock problems and disjunctive scheduling. Their language caters for finite domains and incorporates general control strategies.

So a first task is to identify languages which are instances of the scheme and inherit its semantic properties. This entails the formalization or axiomatization of domains of computation and languages of constraints. Some recent works along that line are now mentioned. Kunen [11] replaces unifiers

by boolean expressions over Herbrand terms which allows the finite representation of negative information. Maher [18] makes a systematic study of this problem extending it to domains of rational and infinite trees. Related work by Mancarella, Martini and Pedreschi [20] is also worth mentioning as well as Wallace's work on negation [25].

We have previously mentioned that we can "lift" Logic Programming by using constraint solving instead of unification. But all the semantics and examples were given related to definite clause programming. This is not what Logic Programming reduces to. Two other main areas are concurrent languages and Data or knowledge bases. Maher has shown in a recent paper [19] that indeed one could also lift aspects of concurrent Logic Programming to handle constraints. Other work in this direction is in progress at Xerox with Saraswat, in Tokyo with Mizoguchi and in New York with McAloon. For data or knowledge bases, it seems clear that the paradigm of constraints in CLP can be pushed further than it has been up to now for languages. Indeed here the full generality of the concept of implicit definitions can be used, in principle a constraint can be "anything" that is a formula as well as a program if one wishes. This is considered by Kannellakis and independently Ramakrishnan who have forthcoming papers that could herald a new and significant area where they analyze and exploit the expressive power of constraints. The work of Imielinski on intelligent queries [5] should also be reexamined in the context of CLP as there are strong conceptual links. If a constraint can be "anything", then it can be a theorem. Indeed Aiba and Sakai [23] in their CAL system which corresponds to an instance of the scheme for non-linear equalities are building a novel form of theorem provers which has great potential. As with all the works previously mentioned the use of constraints raises a number of challenging problems, an important one being the compromise between expressive power and efficiency. There are also aspects of this work which can be relevant to areas outside of Logic Programming. For instance the notion of scheme has been a very powerful tool to analyze semantics properties of languages in our context. But of course this concept can and should be used in other contexts. Constraints are also introduced in rewriting systems. Work by Kapur, in particular jointly with Mohan and Srivas [21] and Comon [2], among others, on systems of equations and inequations and inductive reducibility is clearly relevant and both fields should benefit from a thorough analysis of the connections.

Acknowledgments

We wish to thank C.Lassez and M. Maher for comments on this paper.

Bibliography

1. A. Colmerauer, "Opening the PROLOG-III Universe", Byte Magazine, Special Issue on Logic Programming, August 1987.
2. H. Comon, "Unification et Disunification: Theorie et Applications", PhD Thesis, Grenoble 88.
3. N.C. Heintze, S. Michaylov, P.J. Stuckey, "On the Applications of CLP to Some Problems in Electrical Engineering", Proc. ICLP-4, Melbourne, 1987.
4. N.C. Heintze, J. Jaffar, C.S. Lim, S. Michaylov, P.J. Stuckey, R. Yap, C.N. Yee, "The CLP($\mathscr{R}$) Programmer's manual", Technical Report, Dept. of Computer Science, Monash University, June 1986.
5. T. Imielinski, "Intelligent query answering in rule based systems", Journal of Logic Programming 4(3), Sept 1987.
6. J. Jaffar, J-L. Lassez and M.J. Maher, "A Logic Programming Language Scheme", in Logic Programming: Relations, Functions and Equations", D. DeGroot and G. Lindstrom (Eds), Prentice-Hall, 1985.
7. J. Jaffar and J-L. Lassez, "Constraint Logic Programming", Proc. POPL-87, Munich, 1987.
8. J. Jaffar and J-L. Lassez, "Constraint Logic Programming", Technical report, Department of Computer Science, Monash University, June 1986.
9. J. Jaffar and S. Michaylov, "Methodology and Implementation of a CLP system", Proc. ICLP-4, Melbourne, 1987.
10. J. Jaffar, S. Michaylov, P. Stuckey and R. Yap, "The CLP($\mathscr{R}$) Language and System", To Appear.
11. K. Kunen, "Answer Sets and Negation-as-Failure", Proc. ICLP-4, Melbourne, May 1987.
12. C. Lassez, "Constraint Logic Programming", Byte Magazine, Special Issue on Logic Programming, August 1987.
13. C. Lassez, K. McAloon and R. Yap, "Constraint Logic Programming and Option Trading", IEEE Expert, Special Issue on Financial Software, August 1987.

14. J-L. Lassez and K. Marriott, "Explicit Representation of Terms defined by Counter Examples", Journal of Automated Reasoning, Sept 87 (also IBM Research Report 1986).

15. J-L. Lassez, M. Maher and K. Marriott, "Unification Revisited", to appear in Foundations of Deductive Databases and Logic Programming, J. Minker editor, Morgan Kaufman 1987 (also IBM Research Report 1986).

16. J-L Lassez and K. McAloon, "A Canonical Form for Generalized Linear Arithmetic Constraints", to appear.

17. J-L Lassez, M. Maher and P. Stuckey, In preparation.

18. M. Maher, "Complete axiomatizations of the algebras of finite, rational and infinite trees", Proc. LICS-88, Edinburgh 1988.

19. M. Maher, "Logic Semantics for a Class of Committed-Choice Programs", Proc. ICLP-4, Melbourne, May 1987.

20. P. Mancarella, S. Martini and D. Pedreschi, "Complete Logic Programs with domain closure axioms", To appear in the Journal of Logic Programming.

21. C.K. Mohan, M.K. Srivas and D. Kapur, Forward Reasoning in Systems of Equations and Inequations, to appear, Journal of Logic Programming.

22. K. Mukai and H. Yasukawa, "Complex Indeterminates in PROLOG and its application to discourse models", New Generation Computing 3, 1985.

23. K. Sakai and A. Aiba, "Introduction to CAL", ICOT Memo, 1987

24. P. van Hentenryck and M. Dincbas, "Forward Checking in Logic Programming", Proc. ICLP-4, Melbourne, May 1987.

25. M. Wallace, "Negation by Constraints - a sound and efficient implementation of negation in deductive databases", Proc. SLP-4, San Francisco, 1987.