Constraint Logic Programming

Joxan Jaffar† and Jean-Louis Lassez

Received 10/30/86

I.B.M. Thomas J. Watson Research Center
Yorktown Heights
N.Y. 10598
U.S.A

Abstract

We address the problem of designing programming systems to reason with and about constraints. Taking a logic programming approach, we define a *class* of programming languages, the CLP languages, all of which share the same essential semantic properties. From a conceptual point of view, CLP programs are highly declarative and are soundly based within a unified framework of formal semantics. This framework not only subsumes that of logic programming, but satisfies the core properties of logic programs more naturally. From a user's point of view, CLP programs have great expressive power due to the constraints which they naturally manipulate. Intuition in the reasoning about programs is enhanced as a result of working directly in the intended domain of discourse. This constrasts with working in the Herbrand Universe wherein every semantic object has to be explicitly coded into a Herbrand term; this enforces reasoning at a primitive level. Finally, from an implementor's point of view, CLP systems can be efficient because of the exploitation of constraint solving techniques over specific domains.

† This work was done while J. Jaffar was in the Department of Computer Science, Monash University, Victoria 3168, Australia.

1. Introduction

The problem at hand is that of designing programming systems to reason with and about constraints. Toward this aim, we present the foundations for a class of programming languages based upon constraint solving and the logic programming paradigm. The framework herein, we call it CLP(X), is a scheme in the sense that when one instantiates X with a suitable domain of discourse, one obtains a programming language with several important features. PROLOG, PROLOG-II, PROLOG-III and the CLP system, for example, are instances of the scheme. This scheme may also provide formal semantics for other languages such as the recently proposed CIL [15] and LOGIN [1].

In constraint logic programs, basic components of a problem are stated as *constraints*. the problem as a whole is then represented by putting the various constraints together by means of *rules*. Because unification is but one special case of constraint solving, constraint logic programs have superior expressive power. Furthermore, the programs are naturally amenable to algebraic semantics. Two important points arise here: first, these algebraic semantics form an important complement to the other logic-based semantics. Intuition in the reasoning about programs is enhanced as a result of working directly in the intended domain of discourse. This constrasts with working in the Herbrand Universe wherein every semantic object has to be explicitly coded into a Herbrand term thus enforcing reasoning at a primitive level. Second, an algebraic treatment makes pos-

sible the exploitation of efficient implementation methods from the literature on constraint solving over certain structures. We will illustrate these points using examples drawn from an experimental implementation of a CLP system.

In the next section, we provide the motivations which lead us to design the Constraint Logic Programming Scheme. Following that, we provide examples of programs written for the CLP system, i.e. an experimental implementation of one particular instance of the Scheme, to illustrate the expressive power of CLP programs. We also argue here that efficient implementations of CLP languages can be obtained by the judicious use of special-purpose constraint solvers. Finally, the semantic properties of the Scheme are formally presented. The full treatment of the semantics, including all proofs, is however to be found in [6].

2. Motivation and Background

Much of the present research in Logic Programming concentrates on extensions to Prolog. An important issue is the integration of the essential concepts of functional and logic programming. Another issue is the use of equations to define data types. Recent work along these lines, from Goguen and Meseguer, Kahn, Komorowski, Kornfeld, Reddy, Sato and Sakurai, Subrahmanyam and You will be found in the text edited by DeGroot and Lindstrom [5].

There is some concern that these extensions have little connection left with logic. In fact, the very nature of the concepts in these extensions is such that it is not difficult to accommodate them in standard logic or some variant thereof. The crucial point is not, however, the issue of formalization within or without logic. Rather, it is whether or not the unique semantic properties of logic programs are preserved in the extensions.

Toward this aim, Jaffar, Lassez and Maher [8] proposed a "logic programming language scheme" Its syntax is the syntax of Definite Clauses, its domain of computation is left unspecified but it is assumed to be definable by a unification complete equality theory, and its interpreter is based on SLD resolution and an appropriate generalized unification algorithm. The semantic properties of definite clauses hold for this scheme and all its instances. Now instead of establishing one by one the various semantic results for a given extension to PROLOG, one can use the scheme to obtain them all in one move. This is exemplified in [9] in the case of Colmerauer's PROLOG-II [3] defined over the domain of rational trees. Essentially we proceeded in two steps, first giving an equality theory whose standard model is the intended domain of rational trees, and then showing that this equality theory is unification-complete. Thus PROLOG-II is an instance of this scheme and its semantics is automatically obtained.

While this Scheme achieved the goal of encapsulating, within one unified framework, logic programming languages whose domains are definable by an equality theory, it is intimately tied to, and therefore restricted to, unification. We have shown that the generalisation from standard unification to unification in equality theories posed no conceptual barrier in the semantics of logic programs. When working outside unification and equations, however, it is no longer apparent that a similar scheme exists.

The problem of inequalities in PROLOG-II [4], for example, cannot be accomodated in this Scheme. It was, in fact, not even evident, nor was it to be expected, that the key semantic properties of logic programs still hold here. However, Jaffar and Stuckey [13] showed, from first principles, that these properties do indeed hold.

Consequently, we are now led to consider an extension of the Scheme in which the concept of (generalised) unification is replaced by the concept of constraint solving. Apart from the advantage of being more general, programs in the new Scheme deal with constraints over a given domain of computation, and hence they are provided with *algebraic* semantics. This in turn enhances intuition because reasoning about programs can be done directly within the intended domain using its natural constraints.

3. Programming in an Instance of CLP

In this section we give examples of logic programs with arithmetic constraints in order to give an informal notion of their semantics. This instance of CLP is suitable for numerical applications and operations research. We also wish to highlight the operational differences with traditional logic programming.

We begin with the domain of computation: this is defined to be a two-sorted algebra $\mathcal{A}$ comprising of the natural combination of real arithmetic terms and uninterpreted functors. Only = is used to relate terms which contain uninterpreted functors. For example, X = cons(Y + 3, f(X*Z)) is a valid equation involving two uninterpreted functors cons and f. A *Constraint Logic Program* consists of a finite set of *rules* each being of the form

$$p_0(t_0) :- c_1(u_1), \ \ldots, \ c_n(u_n), p_1(t_1), \ \ldots, p_m(t_m)$$

where p_i, $0 \le i \le m$ are predicate symbols different from the relation symbols in $\mathcal{A}$; c_j, $0 \le j \le n$, are symbols denoting relations in $\mathcal{A}$ (the *constraint* of the rule) and t_k and u_l, $0 \le k \le m$, $0 \le l \le n$, are terms of $\mathcal{A}$ constructed in the obvious manner. A *goal* has the same form as the body of a rule.

The informal declarative semantics is essentially the same as those of standard logic programs. For example,

```
complexmult(c(R1, I1), c(R2, I2) c(R3, I3)) :-
        R3 = R1 * R2 - I1 * I2,
        I3 = R1 * I2 + R2 * I1.
ohmlaw(V, I, R) :- complexmult(I, R, V).
inductorlaw(I, V, L, W) :- complexmult(c(0, W*L), I, V).
capacitorlaw(I, V, C, W) :- complexmult(c(0, W*C), V, I).
```

describes some elementary properties in circuit analysis. Consider the first rule above. The goal

```
?- complexmult(c(R, I), c(10, 50), c(20, 50))
```

entails the solution of the simultaneous equations

$$20 = R * 10 - I * 50$$
$$50 = R * 50 + 10 * I$$

giving the result R = 1.038461 and I = -0.192307.

The informal operational semantics, however, has two main components: a constraint solver for $\mathcal{A}$-relations and an adaptation of the goal-reduction technique in logic programming. A *derivation sequence* consists of goals which have solvable constraints. For example, from the goal

$$?- c(t_1), p(t_2)$$

and the rule

$$p(t_3) :- c2(t_4), q(t_5$$

we may derive the goal

$$?- c(t_1), t_2 = t_3, c2(t_4), q(t_5)$$

providing that the constraint $c(t_1)$ & $t_2 = t_3$ & $c2(t_4)$ is solvable in $\mathcal{A}$

Derivation sequences are *successful* when the last goal therein contains only constraints. These *answer constraints* constitute the output of CLP programs. *Finitely failed* sequences are those whose last goal cannot be expanded.

For example, the following program solves the Dirichlet problem for Laplace's equation in two-dimensions.

```
laplace([H1, H2, H3 | T]):-
   av(H1, H2, H3),
   laplace([H2, H3 | T]).
laplace([__, __]).

av([TL,T,TR | T1], [ML,M,MR | T2], [BL,B,BR | T3]):-
   B + T + ML + MR - 4 * M = 0,
   av([T, TR | T1], [M, MR | T2], [B, BR | T3]).
av([__, __], [__, __], [__, __]).
```

This program outputs a matrix (list of lists) giving the temperature of a surface at discrete points. Typical input is a matrix which contains specific values at the four boundaries.

The program then specifies that the temperature at each non-boundary point is the average of those of four neighbouring points. The goal

```
?- laplace([
  [0,   0,  0,  0,  0],
  [100,  R,  S,  T, 100],
  [100,  U,  V,  W, 100],
  [100,  X,  Y,  Z, 100],
  [100, 100, 100, 100, 100]
 ]).
```

for example, results in the answer

```
laplace([
  [0,      0,     0,     0,  0],
  [100, 57.143, 47.321, 57.143, 100],
  [100, 81.250, 74.999, 81.250, 100],
  [100, 92.857, 90.176, 92.857, 100],
  [100,   100,   100,   100, 100]
 ]).
```

During execution, the PROLOG-like goal reduction technique serves to collect together all the constraints:

```
U + 0 + 100 + S - 4*R = 0,
V + 0 + R + T - 4*S = 0,
...
100 + W + Y + 100 - 4*Z = 0
```

and the second component of the operational model, the constraint solver, then determines the values of the variables R, S, ... , Z.

We may also work with a goal which is "underspecified"; the goal

```
?- laplace([
  [0,  0,  0],
  [100, X, 100],
  [100, B, 100]
 ])
```

results in a *symbolic* answer "X = 50 + 0.25B".

In the final example, we further illustrate the expressive power of using constraints in answers as well as in queries. This example is based upon option pricing.

```
heavyside(X, Y, Z) :- Y < X, Z = 0.
heavyside(X, Y, Z) :- Y ≥ X, Z = 1.
ramp(X, Y, Z) :- Y < X, Z = 0.
ramp(X, Y, Z) :- Y ≥ X, Z = Y - X.

% Option valuation
value(Type, BuyorSell, S, C, P, I, X, B, Value) :-
  Value = H1*T1 + H2*T2 + R1*T3 + R2*T4,
  S ≥ 0, C ≥ 0, P ≥ 0, I ≥ 0, X ≥ 0, B ≥ 0,
  option(Type, BuyorSell, S, C, P, I, X, B,
         K, B1, B2, H1, H2, R1, R2),
  heavyside(B2, S, T1), heavyside(S, B2, T2),
  ramp(B1, S, T3), ramp(B2, S, T4).

% Lookup option vector
option(Type,sell,S,C,P,I,X,B,B1,B2,H1,H2,R1,R2) :-
  table(Type,1,S,C,P,I,X,B,B1,B2,H1,H2,R1,R2).
option(Type,buy,S,C,P,I,X,B,B1,B2,H1,H2,R1,R2) :-
  table(Type,-1,S,C,P,I,X,B,B1,B2,H1,H2,R1,R2).

% M is -1 or 1 depending on buying or selling option
table(stock,M,S,C,P,I,X,B,0,0,M*S*(1+I),  0,-1*M, 0).
table( call,M,S,C,P,I,X,B,0,X,M*C*(1+I),  0,  0,-1*M).
table( put,M,S,C,P,I,X,B,0,X,M*P*(1+I)-X,0,  1,-1).
table( bond,M,S,C,P,I,X,B,0,0,M*B*(1+I),  0,  0, 0).
```

The following straightforward query finds the value of selling a call option given the call price 5, exercise price 50, interest rate 5% and current stock price 60.

```
?- CALL = 5, EX = 50, INT = 0.05, S = 60,
  value(call, sell, S, CALL, __, INT, EX, __, WEALTH)
```

The answer gives WEALTH = -4.75. In the traditional relational programming style, we can turn the question upside down and ask what should be the stock price so that the wealth exceeds 5:

```
?- WEALTH > 5, CALL = 5, EX = 50, INT = 0.05,
   value(call, sell, S, CALL, __, INT, EX, __, WEALTH)
```

Two symbolic answers are obtained:

(a) WEALTH = 5.25 & S < 50,
(b) WEALTH = 55.25 - S & 50 ≤ S < 50.25.

We may also naturally combine options in several ways. For example, a *straddle* (i.e. buying both a put and a call on the same stock with the same exercise price and maturity date) will make money only if the stock price varies significantly. The following goal asks what should the stock price be so that the profit from the straddle is at least 10:

```
?- WEALTH = W1 + W2,
   CALL = 5, PUT = 7, EX = 50, INT = 0.05,
   WEALTH ≥ 10,
   value(call, buy, S, CALL, __, INT, EX, __, W1),
   value(call, sell, S, __, PUT, INT, EX, __, W2).
```

Two answers are obtained:

(a) WEALTH = 37.4 - S & 0 < S ≤ 27.4
(b) WEALTH = S - 62.6 & S ≥ 72.6

which corresponds to the nature of a straddle since we profit by at least 10 if the stock price goes below 27.4 or if it rises above 72.6.

We conclude this section with a few remarks on an experimental implementation: the CLP system [7, 11].

Constraint solvers, such as the simplex algorithm, are aimed at solving a given large set of complex constraints typically arising from Operations Research problems. In principle, such solvers can of course be used in CLP. However, in a programming context, the constraints are obtained incrementally and each increment often involves only relatively simple constraints. There is another important difference. In programs, some constraints are used also in a manner compatible with that of assignments and tests. These, and other special classes of constraints, clearly do not require a general purpose solver.

In short, while traditional solvers attack large and static constraints, ours must deal with *dynamically created smaller constraints which are often of a specialised nature.* These two fundamental differences dictate that a constraint solver used in a programming language be specially engineered.

The present implementation, an interpreter written in C, uses two methods for dealing with the "hard" linear constraints; they are a graph-manipulation procedure and an adaptation of the simplex algorithm. Minimising the frequency of invocation of the general purpose solver is a key factor in its efficiency (the naive list reversal program runs at 10K Lips on a Pyramid 98X). An important issue in the ongoing development of this system concerns the solution of non-linear constraints. Presently, a co-routining mechanism is employed to delay the need for solving such constraints in the hope that further along in a computation sequence, sufficient variables become ground and so the constraints become linear.

4. Semantics for the CLP Scheme

The main results that provide the formal foundations for the logic programming with equality scheme dealt with the following primary concerns:

- *the existence of a canonical domain of computation;*
- *the existence of a least and greatest model semantics;*
- *the existence of a least and greatest fixpoint semantics;*
- *soundness and completeness results for successful derivations of the the underlying implementation model;*
- *soundness and completeness results for finitely failed derivations of the underlying implementation model, and*
- *soundness and completeness results for negation-as-failure.*

Because of the fundamental difference between equality theories and constraint solving, we have to introduce new concepts and techniques in order to obtain corresponding results for CLP. Paramount amongst these are the notions of a solution-compact structure.

We start with a many-sorted structure $\mathcal{A}$ which is the central element in our algebraic semantics. $\mathcal{A}$ is *solution-compact* if (a) every element in $\mathcal{A}$ is the unique solution of a finite or infinite set of constraints (*limit* elements are those which can be defined only by an infinite set of constraints) and (b) every element in the complement of the solution space of a constraint C belongs to the disjoint solution space of some finite or infinite family C_i of constraints. Symbolically, we have, by identifying constraint symbols and their solution spaces below, the following equivalent definition:

(a) $\forall d \in \mathcal{A}, d = \cap C_i$
(b) $\forall C, \overline{C} = \cup C_i$

Thus (a) requires every element to be either finitely definable or a limit element. Condition (b) is very weak. It is met in all plausible structures that we have considered. It is, in fact, necessary and sufficient to establish many results concerning finite failure and negation.

Example 1a. Any structure which has no limit elements is trivially solution-compact. This includes, in particular, any finite structure, the structure $\mathcal{A}_1$ = (HB, =), where HB is some Herbrand universe, and the structure (HB/E, =), where HB/E denotes the quotient of some Herbrand universe by an equality theory E. □

Example 2a. The structure $\mathcal{A}_2$ = (IHB, =, ≠), where IHB is some infinitary Herbrand universe, is used in PROLOG-II. It is easy to check that condition (a) holds. That condition (b) holds follows simply from the fact that the negation of an atomic constraint is itself a constraint. □

Example 3a. Consider the structure $\mathcal{A}_3$ used in the CLP system as described in Section 3. The proof of its solution-compactness can be found in [14]. □

We consider now a correspondence between algebraic and logical semantics. Logic programs enjoy the following property: a goal G is true in all models of P iff G is true in all Herbrand models of P. This allows computation to be performed in the Herbrand universe. For similar reasons, we say that a theory $\mathcal{T}$ *corresponds* to $\mathcal{A}$ if

(i) $\mathcal{A} \models \mathcal{T}$, i.e. $\mathcal{A}$ models $\mathcal{T}$, and
(ii) $\mathcal{A} \models \exists C$ implies $\mathcal{T} \models \exists C$ for all constraints C.

For reasoning about negation, we also require that $\mathcal{T}$ be *satisfaction-complete*:

$\mathcal{T} \models \neg C$ whenever not $\mathcal{T} \models \exists C$.

We remark that this notion corresponds to that of "unification completeness" [8] in the old scheme based on equality theories.

Example 1b. Consider the structure $\mathcal{A}_1$ above, and the theory obtained from the schema

$f(\tilde{x}) \neq g(\tilde{y})$
$z \neq t[z]$
$f(\tilde{x}) = f(\tilde{y}) \rightarrow \tilde{x} = \tilde{y}$

where f and g are distinct functors in the alphabet of $\mathcal{A}_1$ and t[z] is a term containing an occurrence of the variable z.

This theory corresponds to $\mathcal{A}_1$ and is satisfaction-complete. □

Example 2b. Consider the structure $\mathcal{A}_2$ above, and the theory obtained from the schema

$$\forall\tilde{y}\exists!\tilde{x}\ \{\ \tilde{x}_1 = t_1(\tilde{x}, \tilde{y})\ \&\ \ldots\ \&\ \tilde{x}_k = t_k(\tilde{x}, \tilde{y})\ \}$$

where the x_i, $1 \leq i \leq k$, are distinct variables and the t_i, $1 \leq i \leq k$, are terms. This theory corresponds to $\mathcal{A}_2$ but is not, however, satisfaction-complete. To make it so, adjoin the axioms given by the schema

$$f(\tilde{x}) \neq g(\tilde{y})$$
$$f(\tilde{x}) = f(\tilde{y}) \rightarrow \tilde{x} = \tilde{y}$$

where f and g are distinct functors. □

Example 3b. Consider the two-sorted structure $\mathcal{A}_3$ described in section 3 above. Take the theory RCF of real closed fields wherein the variables are universally quantified over the sort of real numbers. Adjoin the theory $\mathcal{T}$ in example 4 wherein all variables are quantified over the remaining sort. The resulting theory corresponds to $\mathcal{A}_3$ and is satisfaction-complete. This is due to a combination of the two reasons: $\mathcal{T}$ is satisfaction-complete and RCF is, simply, complete. □

In order to establish fixpoint semantics, we have, for each CLP program P, the following function mapping from and into $\mathcal{A}$-base (i.e. the cross product between the predicate symbols in P and $\mathcal{A}$).

T(S) = {d ∈ $\mathcal{A}$-base:
there is a rule in P
A :- $\tilde{c}$, $\tilde{B}$
and an $\mathcal{A}$-valuation θ such that
(a) $\mathcal{A} \models (A\theta = d)$,
(b) $\mathcal{A} \models \tilde{c}\theta$, and
(c) $\{B\theta\} \subseteq S$
}

Thus it is a counter-part to the usual function T for logic programs. This function is, however, based upon $\mathcal{A}$-solvability and not unifiability in a Herbrand Universe. Thus if $\mathcal{A}$ were a real arithmetic structure and P contains just the rule

$$p(X) :- 17X^{256} + 35X^{17} - 99X = 0$$

then T({}) = {p(d): d is a zero of the polynomial in P}.

In general, the structure $\mathcal{A}$ has limit elements. We are thus in the new position of having semantic objects for which there is no syntactic counter-part. The problem of referring to these elements arises. In our framework, a limit element is related to syntactic objects, e.g goals, by means of a constraint containing this element in its solution space. This constraint thus represents a finite approximation to the element. Take, for instance the structure and program P immediately above. Let d be a limit element (i.e. a transcendental number) and therefore not a zero of the polynomial. The semantic object p(d) is clearly semantically finitely failed; this can be formalised as $p(d) \notin T(\mathcal{A}\text{-base})$. We can find an interval (r1, r2) such that r1, r2 are rationals, $r1 \leq d \leq r2$ and the goal

$$?- p(X), r1 \leq X, X \leq r2$$

will be finitely failed in one derivation step of the interpreter. In other words, to show that $p(d) \notin T(\mathcal{A}\text{-base})$, we showed that approximation to p(d), which can syntactically presented as a goal, fails.

This technique can be generalised to all solution-compact structures. In order to do so, we introduce a topological notion related to closure: a set $S \subseteq \mathcal{A}$-base has a *finitary cover* if for all $d \in S$, there exists a constraint C such that $d \in C \subseteq S$. We then establish a key lemma which has no counterpart in previous treatments of logic programming. This lemma states that the various sets associated with T have finitary covers:

Finitary Cover Lemma: For all finite n,
(a) $T\uparrow n$ has a finitary cover.
(b) $T\downarrow n$ has a finitary cover.
(c) $\overline{T\uparrow n}$ has a finitary cover.
(d) $\overline{T\downarrow n}$ has a finitary cover. □

This lemma is crucial for proving the soundness and completeness of the interpreter with respect to the semantic notion of finite failure. It is interesting to note that if this result holds for a given structure, then the structure has to be solution-compact.

The only place where there is a discrepancy between the logical and algebraic semantics concerns the operational aspects of finite failure. In the algebraic treatment of CLP, falsity is no longer characterised by the finite failure of the interpreter. This is due to the existence of infinite derivations that give rise to an unsolvable set of answer constraints. The corresponding result is that falsity is characterised by *ground* finite failure, or equivalently, the complement of the greatest fixpoint of T. This discrepancy disappears for *canonical* logic programs, those whose greatest fixpoint is reached in ω steps. This discrepancy also disappears when the structure is compact in the sense that an infinite set of constraints is unsolvable iff some finite subset thereof is unsolvable (e.g PROLOG-II without unequations). Many structures (e.g. PROLOG and the CLP system) are, however, not compact.

It follows that the completeness of the negation-as-failure rule does not hold, in general, in the algebraic semantics. In order to resolve this problem, we extend, in a more natural setting, the discussion in [10] and the result of [12] which showed that every logic program is equivalent to a canonical one.

Summary of Main Results

The results we present here provide a formal semantics for CLP languages. They are comprehensive in the sense that they cover algebraic, fixpoint, model-theoretic and operational aspects. The fundamental results concerning logic programs, as described in [10], have their counterparts here.

Theorem 1.
Least model of P = Least fixpoint of T = [SS(P)].

Theorem 2. $\overline{T\downarrow\omega} = [FF(P)]$.

Theorem 3. $\overline{gfp(T)} = [Ground\_\_FF(P)]$.

Theorem 4. (Negation-as-Failure, Logical Version)
$\mathcal{I} \,\&\, P \models \neg G$ iff $G \in FF(P)$.

Theorem 5. (Negation-as-Failure, Algebraic Version)
The following statements are equivalent:
(a) P is canonical.
(b) $\mathcal{A} \,\&\, P \models \neg G$ iff $G \in FF(P)$.

5. Conclusion

There are languages involving constraints such as [2] and [16]. These, as well as many others concentrated on extending PROLOG, are typically motivated by implementation issues or specific applications. Our approach in this paper is fundamentally different because we establish *first* a formal framework with pre-defined semantic properties. This is then used to define languages that cater for different applications.

All the languages in this class share the same essential semantic properties by being encapsulated in one unified framework of formal semantics. This framework is not just more general than that of logic programming, but satisfies the core properties in a more natural setting. The resulting CLP languages satisfy the three criteria for a programming language in that they are soundly based, they have substantial expressive power, and that efficient implementations can be constructed.

Acknowledgements

We are grateful to P.J. Stuckey for valuable discussions during the early part of this work. S. Michaylov was instrumental in the construction of the CLP solver and the development of a programming methodology. Many thanks due to R. Yap, N. Heintze, C.S. Lim and C. Yee for their various contributions to the CLP system.

References

[1] H. Ait-Kaci and R. Nasr, LOGIN: A Logic Programming Language with Built-In Inheritance, *Journal of Logic Programming*, 3(3), 1986.

[2] A. Borning, THINGLAB - A Constraint Orientated Simulation Laboratory, *ACM TOPLAS*, 3(4), 1981.

[3] A. Colmerauer, Prolog and Infinite Trees, in: *Logic Programming*, K.L. Clark and S.A. Tarnlund (Eds.), Academic Press, New York, 1982.

[4] A. Colmerauer, Solving Equations and Inequations on Finite and Infinite Trees, *Proc. Conference on Fifth Generation Computer Systems*, Tokyo, November 1984.

[5] D. DeGroot and G. Lindstrom (eds.), *Logic Programming: Relations, Functions and Equations*, Prentice Hall, 1986.

[6] J. Jaffar and J-L. Lassez, Constraint Logic Programming, Technical Report, Department of Computer Science, Monash University, June 1986.

[7] N.C. Heintze, J. Jaffar, C.S. Lim, S. Michaylov, P.J. Stuckey, R. Yap and C.N. Yee, The CLP Programmer's Manual, Department of Computer Science, Monash University, June 1986.

[8] J. Jaffar, J-L. Lassez and M.J. Maher, A Logic Programming Language Scheme, in: *Logic Programming: Relations, Functions and Equations*, D. DeGroot, G. Lindstrom (eds.), Prentice Hall, 1986.

[9] J. Jaffar, J-L. Lassez and M.J. Maher, PROLOG-II as an Instance of the Logic Programming Language Scheme, in *Formal Descriptions of Programming Concepts*, M. Wirsing (Ed), North-Holland, 1986.

[10] J. Jaffar, J-L. Lassez and M.J. Maher, Issues and Trends in the Semantics of Logic Programming, *Proc. 3rd International Conference on Logic Programming*, London, July 1986.

[11] J. Jaffar and S. Michaylov, Methodology and Implementation of a Constraint Logic Programming System, Technical Report, Computer Science Dept., Monash University, June 1986.

[12] J. Jaffar and P.J. Stuckey, Canonical Logic Programs, *Journal of Logic Programming*, 3(2), 1986.

[13] J. Jaffar and P.J. Stuckey, Semantics of Infinite Tree Logic Programming, *Theoretical Computer Science*, to appear.

[14] J. Jaffar and P.J. Stuckey, A Separation Algorithm for Theories with Uninterpreted Functors, Forthcoming.

[15] K. Mukai and H. Yasukawa, Complex Indeterminates in PROLOG and its Application to Discourse Models, *New Generation Computing*, 3, 1985.

[16] G.L. Steele, The Definition and Implementation of a Computer Programming Language based on Constraints, Ph.D. Thesis, M.I.T. AI-TR 595, 1980.