

Some Issues and Trends in the Semantics of Logic Programming

J. Jaffar, J-L. Lassez, M.J. Maher
IBM Thomas J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

A major problem facing the designers of programming languages is the conflict between expressive power (or “high-levelness”) and efficiency of execution. In the conventional approach expressive power has been consistently sacrificed in an ad hoc manner to efficiency and often has been confused with the accumulation of “useful” features. This has resulted in the design of intricate languages such as ADA. Both Backus and Hoare in their Turing award addresses have been highly critical of this approach. The hope that the complexity of such languages could be mastered by using the sophisticated theoretical tools of denotational semantics has not been fulfilled.

Logic-based programming languages, on the other hand, have the great advantage that formal tools such as models and resolution used to capture their semantic properties do so in a simple and natural manner.

In the first part we will review the main semantic properties of definite clauses, which form a theoretical basis for the study of PROLOG. These are model theoretic semantics, least fixedpoint semantics, finite failure and negation as failure. We will mention forthcoming results and discuss some remaining open problems. We assume some familiarity with the literature, and refer the reader to Lloyd’s account [25] for the basic results and terminology. In a second part we consider extensions to definite clause logic programs. Our concern is to characterize extensions which still possess the unique semantic properties that are discussed in the first part. These properties do not depend on the Herbrand universe and standard unification, but on their abstract relationship. Any domain and unification theory which satisfy this relationship can be used without incurring any loss in semantic properties. This approach is then substantially extended by allowing constraints over user-oriented domains. This results in a formalism which brings the theory of logic programming closer to standard programming practice.

Definite clause logic programs

In 1879 Frege invented a system of symbolic representation, manipulation and computation of “pure thought” which he called BEGRIFFSCHRIFT. This system is better known in English under the name of Predicate Calculus. The two relevant aspects of Predicate logic are model theory and proof theory: model theory corresponds to specification and declarative notions, proof theory corresponds to operational semantics and implementations. In other words model theory is used to formalize “what” we want to be computed and proof theory is used to formalize “how” to compute it.

The meaning we give to a program, *as a logic formula*, is the set of logical consequences, that is the set of formulas which are true in every model of the program. The meaning we give to a program, *as a program*, (rather than as a logic formula) is the set of formulas which are true in all models of the program which use an intended domain and an intended interpretation of the predicate and function symbols. The fact that logical consequence does not depend on the interpretation given to the various symbols allows the intended interpretation to be factored out and the computation can be safely performed in a purely symbolic context.

The best known example is within clausal form predicate logic, and devised by Robinson. Using unification and resolution to manipulate the symbols of the Herbrand universe and the Herbrand base, the logical consequences of a program can be symbolically computed in a sound and complete manner. When the computation finds that A is true in all Herbrand (that is symbolic) models it follows that A is true in all models and a fortiori in the model intended by the person who wrote the program. It should be noted here that this very strong property might be more than what we require. The Skolem-Lowenheim theorem shows that we can find an arbitrary number of arbitrarily complex models for our program. To know that A will also be true in models which are artificial, arbitrary and altogether irrelevant to our problem, is neither required nor useful. This prompts the idea of the algebraic approach, which we will mention later, in which a single domain is considered, in contrast with the logic approach which allows a multiplicity of domains.

However speeds of execution that are expected of programming languages have not been achieved in the general framework of clausal form. The restriction to definite clauses, which leads to a drastic loss in expressive power, has nevertheless distinct advantages: computationally, the combination of definite clauses and the specialization of Kowalski and Kuehner’s SL resolution [18] to definite clauses (called LUSH resolution by Hill [11] and

SLD resolution by Apt and van Emden [1]), is sufficiently efficient to form a basis for the implementation of a programming language; logically, SLD resolution is sound and complete for atomic logical consequences; and definite clauses retain the ability to represent all computable functions as shown by Tarnlund [40], Sebelik and Stepanek [37]. We will review now some key semantic properties of definite clause logic programs.

Model theoretic semantics

As we observed above, we can restrict our attention to Herbrand models. For brevity, throughout this section we will simply say “model” when referring to Herbrand models and we will assume that A is a ground atom. For A to be a logical consequence means that A belongs to all models. In general the intersection of models is not a model so no model can be selected as fully representative of the notion of logical consequence. However, in the case of definite clauses, the intersection of all models is a model, as pointed out by Van Emden and Kowalski [7]. Consequently the notion of ground atomic logical consequence can be captured by a single model, namely the intersection of all models, which is the least one.

However one must be careful when using this least model approach: although the assignment of true to an atom A corresponds to A being a logical consequence, the assignment of false never corresponds to $\neg A$ being a logical consequence. The approach makes sense only in conjunction with the closed world assumption, where all ground atoms not in the least model are assumed to be false. But this implies incompleteness: some atoms will lead to infinite computations and the interpreter will not be able to determine their truth value.

An alternative approach, has been proposed by Lassez and Maher [23] whereby an atom is assigned true (false) if and only if it is true (false) in all models. Otherwise there is not enough information in the program to assign a truth value to an atom, it will be true in some models and false in others. This ambiguity is reflected by the assignment of an undefined truth value to the atom. Consequently we can have completeness, as atoms which lead to infinite computations are assigned the value undefined. This approach has been pursued recently by Fitting [9] and is based on work of Manna and Shamir [28, 29, 30], Kripke [20] and old ideas of Herbrand, Godel and Buchi [2]. Together with Mycroft [31] we believe that the notion of undefined should be part of any declarative or operational semantics of programming languages, whether based on logic or not.

Fixedpoint semantics

As narrated by Lassez, Nguyen and Sonenberg [24] there is a fair amount of confusion concerning fixedpoint semantics in the literature. Fortunately in the case of logic programs the situation is very clear and simple. Consider the informal semantics of a program, “You have facts and you have rules. Apply the rules to the facts to generate new facts. Repeat this operation until you cannot generate new facts.” This amounts to define a set inductively. If a function represents the application of the rules to transform a old set of facts into a new set of facts, we have a fixedpoint when the new set of facts is equal to the preceding set of facts. The fixedpoint constructed that way happens to be the least.

This is a very general phenomenon, but the choice of the least fixedpoint is, from a declarative point of view, somewhat arbitrary or inadequate, as argued in previously mentioned references [28, 29, 30]. The fact that we have a least fixedpoint simply means that we have abstracted a computational process, by itself it does not provide a strongly motivated declarative semantics. The fixedpoint which, according to Manna and Shamir, is the most “natural” is called the optimal fixedpoint and is, in general, not equal to the least one. One advantage of logic programs over conventional programs is that the least fixedpoint is equal to the least model, as shown by Van Emden and Kowalski [7], therefore it is associated to logical consequence and has a meaningful declarative interpretation.

In the context of three valued logic, that is when undefined is assigned to literals which are not logical consequences, the situation is even more striking. Lassez and Maher [23] have shown that the notions of models and fixedpoints coincide. Models are used to give semantics to logic formulas, fixedpoints are used to give semantics to recursive definitions. As a program can be viewed both as a logic formula and as a recursive definition, it is more satisfactory to have the two notions of model and fixedpoint coincide, rather than having the equality only in the case of least elements. Now we still have that the least model captures the notion of logical consequence and is the right choice to give a declarative semantics based on logic. Furthermore it is equal to the optimal fixedpoint which is the right choice to give a declarative semantics to a recursive definition. As models and fixedpoints coincide, the optimal fixedpoint is equal to the least fixedpoint. There is no need to argue over which fixedpoint semantics is more desirable.

Further fixedpoint semantics have been given by Lassez and Maher [21, 22] where definite clauses are viewed as a production system. Here again the notions of fixedpoint and model coincide. This view also allows the

establishment of a relationship between the semantics of the rules and the semantics of the program, in other words it provides an elementary denotational semantics. This semantics was used to study the decomposition of a program into an equivalent collection of subprograms which can be run independently. It is also suitable for the study of the dual problem of composition of modules to form whole programs. This was shown by O’Keefe [34] who essentially used this semantics to formalize a notion of modularity.

Finite Failure

By definition, when we compute partial recursive functions, there are inputs which lead to infinite computations. As Kowalski pointed out [19], in the context of logic programming there is no limit to the amount of infinite branches that can be pruned from the search space (by loop-checking methods etc.), and no algorithm can exist to eliminate all of them. Consequently we can further and further approximate the closed world assumption but never, in general, reach it.

If we cannot eliminate all infinite computations which are due to the nature of the problem, we should at least do our best not to introduce new ones via our interpreter. In that respect resolution behaves badly - a large number of infinite computations that have nothing to do with the problem to be solved are nevertheless introduced.

Apt and Van Emden [1] introduced fixedpoint techniques to establish in a more elegant way various results, for instance Hill’s soundness and completeness of SLD resolution [11]. Their methods have become standard since. They met, however, with difficulties when characterizing the finite failure of SLD resolution to prove an atom to be a logical consequence. This led to long and involved proofs and a weak result, difficult to interpret semantically: the SLD finite failure set is equal to the complement of $T \downarrow \omega$. The result is difficult to interpret semantically, as the complement of $T \downarrow \omega$ is an abstract mathematical object implicitly defined; we just have a purely mathematical characterization of the SLD finite failure set. It is weak in the sense that $T \downarrow \omega$ only implies that there exists a finitely failed SLD tree for A while infinite SLD trees for A may also be present. If we wish to make use of this result, we must build an interpreter which generates all SLD trees, a fairly daunting prospect.

Lassez and Maher [22] made the remark that the standard concept of fairness should be introduced in SLD resolution in order to remove a class of infinite computations due to the interpreter’s design. The result was startling; most difficulties in the treatment of SLD finite failure were due to

the presence of these unnecessary loops. With this modification, all trees (for a given goal) behave the same way: either all fail or none fail. Furthermore, if some SLD tree for A is finitely failed then all fair SLD trees for A are finitely failed. Consequently, if we have a fair implementation of SLD resolution, testing for SLD finite failure simply requires the generation of a single tree, as is the case for success.

Moreover, a general definition of the finite failure set FF was provided, independent of any implementation (SLD or otherwise), which was clearly needed and simple to derive. It turned out that $A \in FF$ can be easily reformulated into $A \notin T \downarrow \omega$, so this abstract mathematical formula has, in fact, a meaning: it defines the general notion of finite failure! Consequently one could derive the soundness and strong completeness of fair SLD resolution with respect to finite failure and the soundness and weak completeness of SLD resolution with respect to finite failure. These results provided an appropriate setting in which to address the problem of negation as failure. We conclude this section on finite failure with a brief presentation of a topic of current research.

An approach to increasing the finite failure set, while preserving the success set has been proposed by Naish and Lassez [33]. For any program P , the set of instances of P (that is programs more specific than P) which have the same set of successful ground derivations has a least element, called a most specific logic program. This most specific logic program has not only a larger finite failure set than P but its SLD trees are shorter than those of P . There is no known algorithm to transform a program into its most specific version and we strongly suspect that there is none. However some heuristics are developed, which show that this most specific version of a program can be obtained in a significant number of cases.

Negation as Failure

Clark provided a formal framework to study negation as failure [3], via the notion of complete logic programs, and proved the soundness of this rule. This was quite important as it provided a better understanding of the negation as failure rule and opened an active area of research. But also it was very nice technically as, complete programs not being in clausal form, it was not evident that resolution could still be meaningful.

Apt and Van Emden [1] used their fixedpoint techniques to formalize negation as failure. However, they worked in the restricted case of the Herbrand Universe and syntactic identity as an equality theory, so their result of soundness was weaker than Clark's. The time was ripe to put together

Clark's formalism, Apt and Van Emden's fixedpoint techniques, Lassez and Maher's results on the soundness and completeness of finite failure. This is what Jaffar, Lassez and Lloyd did in [13] where they established the completeness of the negation as failure rule with a fairly involved proof, which has since been presented in many different ways.

We present now a different point of view for the negation as failure rule, which relates it to standard theorem proving techniques. This will be done in the propositional case and illustrated by the following simple example

$$\begin{aligned}
 A &\leftrightarrow (B \wedge C) \vee D \\
 B &\leftrightarrow E \vee F \\
 C \\
 \neg D \\
 \neg E \\
 \neg F
 \end{aligned}$$

IFF

We can rewrite this complete program in clausal form by considering separately the if and only-if parts of IFF. It is well-known that a complete program has an if part which is made up of definite clauses IF and that an atom is made true by IFF exactly when it is made true by IF. In this case the only-if part can also take a form similar to definite clauses FI *negated* atoms appear in the head and the body.

$$\begin{array}{ll}
 A \leftarrow B, C & \neg A \leftarrow \neg B, \neg D \\
 A \leftarrow D & \neg A \leftarrow \neg C, \neg D \\
 B \leftarrow E & \neg B \leftarrow \neg E, \neg F \\
 B \leftarrow F & \neg D \\
 C & \neg E \\
 & \neg F
 \end{array}$$

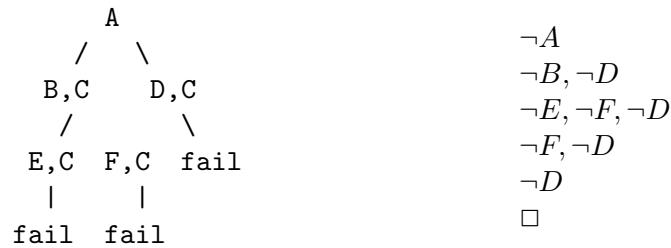
IF

FI

The duality between IF and FI enables us to characterize the atoms made false by IFF using the previous result: an atom is made false by IFF exactly when it is made false by FI.

Clearly this result enables us to optimize a resolution theorem-prover by using SLD resolution on just the IF part of IFF, if we have an atom to prove, and using SLD resolution on just the FI part if we have a negated atom to prove. The optimization comes from the fact that SLD resolution can be

used, useless interactions between the IF and FI parts are automatically avoided. More surprising is the fact that this is, essentially, what is done when the negation-as-failure rule is used with SLD resolution on IF. To make this point clearer, consider the following finitely failed SLD tree for A using IF as the program and the successful SLD derivation for $\neg A$ using FI as the program.



Here we can see that the goals in the derivation correspond to cross-sections of the branches of the failed SLD tree, each atom in a goal corresponding to a single failed subtree. SLD resolution on an atom in the derivation corresponds to considering all the immediate subtrees of the goal containing that atom in the failed SLD tree. Atoms, such as C , which do not contribute to failure in the SLD tree are ignored in the derivation.

In general the relation between a finitely failed tree for the IF part and a successful SLD derivation for the FI part is not so obvious. In particular, it is necessary to Skolemize the existential variables in the FI part of IFF and to include some form of Clark's equality axioms. A demonstration that every false atom has a proof from IFF, in a conventional proof system, that corresponds to a finitely failed tree of the IF part, would constitute a more direct proof of the completeness of the negation as failure rule. If the proof system were similar to SLD resolution then the demonstration would be a step towards an incremental implementation of negation in a manner similar to the work of Sato and Tamaki [35], Schultz [36] and Naish [32].

The Asymmetric Theory

The formal semantics of complete logic programs are not well balanced. We would like a duality between True and False, success and failure, least

fixedpoint and greatest fixedpoint, least model and greatest model such that:

$$\begin{array}{cc}
 A \text{ is True} & A \text{ is False} \\
 \text{iff} & \text{iff} \\
 A \in \text{least (Herbrand) model} & A \notin \text{greatest (Herbrand) model} \\
 \\
 A \in SS & A \in FF \\
 \text{iff} & \text{iff} \\
 \text{There exists a successful} & \text{Every ground derivation is} \\
 \text{ground derivation} & \text{finitely failed} \\
 \\
 T \uparrow \omega = lfp(T) & T \downarrow \omega = gfp(T)
 \end{array}$$

As we pointed out earlier A True implies that A is true in all models, not just the models over an intended domain of interpretation, here the Herbrand Universe. So we would also like to have the duality:

$$\begin{array}{cc}
 A \text{ is true in all models} & A \text{ is false in all models} \\
 \text{iff} & \text{iff} \\
 A \text{ is true in all} & A \text{ is false in all} \\
 \text{Herbrand models} & \text{Herbrand models}
 \end{array}$$

This duality would imply that the logic approach and the algebraic approach coincide. Unfortunately, if all the statements on the left hand side hold, none of the statements on the right hand side hold. The reason lies in the discrepancy between two notions of finite failure:

The first notion is the one introduced by Lassez and Maher [22] which can be worded in the following way

$$A \in FF \text{ iff there exists } n \text{ such that all ground derivations for } A \text{ are failed with length } \leq n$$

This set FF is computable via fair SLD resolution.

The second notion was introduced in Jaffar, Lassez and Maher [15]. It is called ground finite failure:

$$A \in GFF \text{ iff all ground derivations for } A \text{ are failed finitely.}$$

The difference with FF is that even though all derivations are finitely failed, their length is not bounded, and most importantly the set GFF is not computable.

We compute FF , but it is the non computable GFF which is characterized via the greatest fixedpoint

$$FF \subseteq GFF = \overline{gfp(T)}$$

If $A \in GFF$ and A is not in FF , then there exists an infinite SLD derivation for A which cannot be grounded. So the interpreter is manipulating symbols that cannot be bound to elements of the intended domain, leading to the discrepancy between the logic and algebraic approaches. This also implies that in the algebraic approach the negation as failure rule is not complete.

However all the right hand side results hold, and we have a symmetric theory, if we restrict ourselves to programs such that $FF = GFF$ or, equivalently, $T \downarrow \omega$ is equal to the greatest fixedpoint. From a purely mathematical point of view this is a rather exceptional case. However, it appeared that even though it is easy to construct programs such that $T \downarrow \omega$ is not equal to the greatest fixedpoint they are all very contrived and do not correspond to any program arising from standard practice. Thus we were led to conjecture that all “decent” programs satisfy $T \downarrow \omega = GFP(T)$. If this informal conjecture holds then we have a much simpler and elegant theory for logic programs. An important step in that direction was achieved by Jaffar and Stuckey [17]. Calling “canonical” the programs such $T \downarrow \omega = GFP(T)$, they show that every logic program is equivalent to a canonical program. Therefore the class of canonical programs is representative of the class of all programs.

In a similar attempt to define a class of programs with good semantic properties, Fitting [9] recently proposed that the only acceptable programs were those satisfying a condition which, for definite clauses, is equivalent to requiring T to be down-continuous. Down-continuity implies that $T \downarrow \omega = GFP(T)$, so the “acceptable” programs form a subclass of the “decent” programs. However Maher’s characterization of down-continuous programs [26, 27] shows that the “acceptable” program class is too small - “acceptable” programs cannot express transitive closure, for example.

Conservative Extensions

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 [6].

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 we want to address is not the issue of formalization within or without logic, but whether the unique semantic properties of logic programs are preserved in the extensions.

For instance, Hansson and Haridi [10] and, independently, van Emden and Lloyd [8] re-interpreted PROLOG II in standard logic but did not address the key semantic issues: establish the existence of least model and least fixedpoint semantics and the corresponding soundness and completeness results for successful derivations, establish also the soundness and completeness results for finite failure and for the negation as failure rule. In fact what we require is that all the basic theory for definite clauses be rewritten for PROLOG II. This could represent a major undertaking, which should be repeated for any extension or modification to PROLOG. We will describe now a comparatively simple method to solve that problem.

The Scheme

In Jaffar, Lassez and Maher [15] we proposed a “logic programming language scheme” that is now briefly explained. Consider the language of definite clauses as being defined with the following components: The syntax of definite clauses, the Herbrand universe as the domain of symbolic computation together with syntactic identity as underlying equality theory, an interpreter based on fair SLD resolution, a unification algorithm, and negation as failure.

Many extensions can be viewed or formalized by replacing the domain of symbolic computation by another which is obtained by replacing syntactic identity by another equality theory. We would like to preserve the key property of the Herbrand universe, namely that we have a single domain in which logical consequence can be determined. This property will hold if and only if the equality theory has a finest congruence. In such a case one can easily derive least model and least fixedpoint semantics similar to those of definite clauses. This is not a contrived case as Horn equality theories admit finest congruences, and represent a large and useful class of equality theories, they include in particular equational theories. It then appears that the declarative properties of Definite Clauses over the Herbrand Universe

are shared by Definite Clauses over a large class of domains which naturally generalize the Herbrand Universe. The operational aspects of success and failure can be adapted by considering generalized unification and preserving SLD resolution. The corresponding results of soundness and completeness follow naturally.

As for the standard case negation as failure requires more attention. For an equality theory E , terms s and t are unifiable iff $E \models \exists x(s = t)$. With the negation issue at hand we need a dual property; that is we need to establish a relationship between non existence of E unifiers and falsity in E . Informally we say that E is unification complete if every possible solution of a given equation can be represented by an E unifier of the equation. In particular when there are no E unifiers there can be no solution. The soundness and completeness of the negation as failure rule holds for unification complete equality theories.

Consequently we have established the existence of 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, its interpreter is based on SLD resolution and an appropriate generalized unification algorithm. The semantic properties of definite clauses hold for the 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 Jaffar, Lassez and Maher [16] in the case of Colmerauer's PROLOG II [4], defined over the domain of rational trees. Essentially we proceed in two steps, first give an equality theory whose standard model is the intended domain of rational trees, then show that this equality theory is unification complete. PROLOG II can be viewed as an instance of the scheme and possesses its semantics properties. We therefore have a logical interpretation of Colmerauer's rewriting system. This treatment does not address the problem of inequalities in PROLOG II [5]. It is done in the next section which considers constraints in Definite Clauses.

Constraints

The idea of the scheme defining a class of languages which share the same abstract semantic properties is repeated here. Instead of mapping the intended domain on the Herbrand Universe and using specialized unification, programming is done directly in the intended domain using its natural constraints. We thus use an algebraic framework as well as a logic programming one. This revision of the scheme is called CLP which stands for Constraint

Logic Programming.

A CLP program consists of constrained rules which are of the form

$$A \leftarrow c_1, c_2, \dots, c_k \ \square \ B_1, B_2, \dots, B_n$$

where A, B_1, B_2, \dots, B_n are atoms and c_1, c_2, \dots, c_k form a set of constraints. A goal is of the form

$$\leftarrow c_1, c_2, \dots, c_k \ \square \ B_1, B_2, \dots, B_n$$

The interpreter of an instance of CLP consists of the standard goal reduction technique of logic programming and a constraint solver. Implementations of this model of computation can benefit from the wide literature on the problem of constraint solving.

Allowing constraints in the goals and bodies considerably raises the expressive power. A comparison can be made with the introduction of negated atoms in the body of clauses of definite clause programs. It is well known (Clark [3], Shepherdson [38,39]) that the implementation of negation via variants of SLD resolution and negation as failure works only in restricted cases. Furthermore, the semantic properties of definite clauses that we have described no longer hold. It is therefore quite significant to note that CLP programs have least fixedpoint and least model semantics, and results of soundness and completeness that are similar to those of definite clause logic programs, despite the introduction of constraints. Furthermore, the algebraic and logical approaches also coincide for canonical programs. A formal presentation of these results is to be found in Jaffar and Lassez [12].

Colmerauer is now working on Prolog III and it seems that, as in the case of Prolog II and the scheme, we will be able to show that Prolog III can be viewed as an instance of CLP given the formal definition of Prolog III.

Conclusion

The simplicity and elegance of definite clauses makes this formalism attractive from a theoretical point of view. The objects in this formalism are the uninterpreted terms over the Herbrand universe. Programming however is not done exclusively in the Herbrand universe, but uses higher level concepts such as arithmetic. In that sense we can view definite clauses as the Turing machines of Logic Programming. This gap between theory and programming practice can be reduced by introducing user-oriented domains

into the formalism. We have seen that this can be achieved without losing the important properties of definite clauses.

Acknowledgements: The support from IBM Australia is gratefully acknowledged and we thank the text consultants at Yorktown Heights for their advice.

References

- [1] K.R. Apt and M.H. van Emden, Contributions to the Theory of Logic Programming, *Journal of the ACM* 29, 3 (1982), 841-862.
- [2] R. Buchi, Private communication.
- [3] K.L. Clark, Negation as Failure, in: *Logic and Databases*, H. Gallaire, J. Minker (eds.), Plenum Press, 1978.
- [4] A. Colmerauer, Prolog and Infinite Trees, in: *Logic Programming*, K.L. Clark and S.A. Tarnlund (eds.), Academic Press, New York, 1982.
- [5] A. Colmerauer, Solving Equations and Inequations on Finite and Infinite Trees, *Proc. Conference on Fifth Generation Computer Systems*, Tokyo, November 1984.
- [6] D. DeGroot and G. Lindstrom (eds.), *Logic Programming: Relations, Functions and Equations*, Prentice Hall, 1986.
- [7] M.H. van Emden and R.A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *Journal of the ACM* 23,4 (1976), 733-742.
- [8] M.H. van Emden and J.W. Lloyd, A Logical Reconstruction of Prolog II, *Proc. 2nd. Conference on Logic Programming*, Uppsala, Sweden, 1984, 35-40.
- [9] M. Fitting, A Kripke-Kleene Semantics for Logic Programs, *Journal of Logic Programming* 2, 4 (1985), 295-312.
- [10] A. Hansson and S. Haridi, Programming in a Natural Deduction Framework, *Proc. Conference on Functional Languages and their Implications for Computer Architecture*, Goteborg, Sweden, 1981.
- [11] D. Hill, LUSH-resolution and its Completeness, DCS Memo 78, Dept. of Artificial Intelligence, University of Edinburgh, 1974.
- [12] J. Jaffar and J-L. Lassez, Constraint Logic Programming, forthcoming
- [13] J. Jaffar, J-L. Lassez and J.W. Lloyd, Completeness of the Negation-as-Failure Rule, *Proc. 8th. IJCAI*, Karlsruhe, 1983, 500-506.

- [14] J. Jaffar, J-L. Lassez and M.J. Maher, A Theory of Complete Logic Programs With Equality, Proc. Conference on Fifth Generation Computer Systems, Tokyo, November 1984, 175-184.
- [15] 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. Also Technical Report TR 84/15, University of Melbourne, 1984.
- [16] J. Jaffar, J-L. Lassez and M.J. Maher, Prolog II as an Instance of the Logic Programming Language Scheme, Technical Report, Monash University, 1984.
- [17] J. Jaffar and P.J. Stuckey, Canonical Logic Programs, Journal of Logic Programming, to appear
- [18] R.A. Kowalski and D. Kuehner, Linear Resolution with Selector Function, Artificial Intelligence 2, (1971), 227-260.
- [19] R.A. Kowalski, Logic for Problem Solving, North Holland, New York, 1979.
- [20] S. Kripke, Outline of a Theory of Truth, Journal of Philosophy 72 (1975), 690-716.
- [21] J-L. Lassez and M.J. Maher, The Denotational Semantics of Horn Clauses as a Production System, Proc. National Conference on Artificial Intelligence (AAAI-83), Washington D.C., August 1983, 229-231.
- [22] J-L. Lassez and M.J. Maher, Closures and Fairness in the Semantics of Programming Logic, Theoretical Computer Science 29, (1984), 167-184.
- [23] J-L. Lassez and M.J. Maher, Optimal Fixedpoints of Logic Programs, Theoretical Computer Science 39, (1985), 15-25.
- [24] J-L. Lassez, V. Nguyen and E.A. Sonenberg, Fixed Point Theorems and Semantics: A Folk Tale, Information Processing Letters 14, 3 (1982), 112-116.
- [25] J.W. Lloyd, Foundations Of Logic Programming, Springer-Verlag, 1984.
- [26] M.J. Maher, Semantics of Logic Programs, Ph.D. dissertation, University of Melbourne, 1985.
- [27] M.J. Maher, Equivalences of Logic Programs, Proc. 3rd. Logic Programming Conference, London, 1986.
- [28] Z. Manna and A. Shamir, The Theoretical Aspect of the Optimal Fixed Point, SIAM Journal on Computing 5 (1976), 414-426.
- [29] Z. Manna and A. Shamir, The Optimal Approach to Recursive Programs, Communications of the ACM 20 (1977), 824-831.
- [30] Z. Manna and A. Shamir, A New Approach to Recursive Programs, in: Perspectives on Computer Science, A.K. Jones (ed.), Academic Press,

New York, 1977.

[31] A. Mycroft, Logic Programs and Many-valued Logic, Proc. 1984 Symposium on Theoretical Aspects of Computer Science, M. Fontet and K. Mehlhorn (eds.), Springer Lecture Notes in Computer Science 166, 274-286.

[32] L. Naish, Negation and Control in PROLOG, Ph.D. dissertation, University of Melbourne, 1985.

[33] L. Naish and J-L. Lassez, Most Specific Logic Programs, Technical Report, Dept. of Computer Science, University of Melbourne, 1984.

[34] R.A. O'Keefe, Towards an Algebra for Constructing Logic Programs, Proc. Symposium on Logic Programming, Boston, 1985.

[35] T. Sato and H. Tamaki, Transformational Logic Program Synthesis, Proc. Conference on Fifth Generation Computer Systems, Tokyo, 1984.

[36] J.W. Schultz, The Use of First-Order Predicate Calculus as a Logic Programming System, M.Sc. dissertation, University of Melbourne, 1984.

[37] J. Sebelik and P. Stepanek, Horn Clause Programs Suggested by Recursive Function, Logic Programming, K. L. Clark and S.A. Tarnlund (eds.), Academic Press, New York, 1982.

[38] J.C. Shepherdson, Negation as Failure: A Comparison of Clark's Completed Data Base and Reiter's Closed World Assumption, Journal of Logic Programming 1, 1 (1984), 51-79.

[39] J.C. Shepherdson, Negation as Failure II, Journal of Logic Programming 2, 3 (1985), 185-202.

[40] S.A. Tarnlund, Horn Clause Computability, BIT 17, 2 (1977), 215-226.

Note: This version of the paper has been reconstituted with optical character recognition from hard copy. It may contain errors because of this process. In addition, the formatting of this paper is different from the published version.