2000

A Bird's Eye View of the Stable Model Languages

James D. Jones
University of Arkansas at Little Rock, james.d.jones@acm.org

A BIRD'S-EYE VIEW OF THE STABLE MODEL LANGUAGES

James D. Jones

Computer Science
College of Information Science and Systems Engineering
University of Arkansas at Little Rock
Little Rock, AR. 72204-1099
james.d.jones@acm.org

I. Abstract

Logic programming presents us with a wonderful paradigm within which to develop reasoning systems. This paradigm is very expressive, has well understood mathematical properties, and is an area of intense international research. However, the research has not yet been adopted by the practitioner community. Current implementations center around rule-based systems, and object-oriented systems. Thus, the practitioner community is missing out on very powerful reasoning tools.

The semantics of logic programs is a very difficult, technical arena. The purpose of this paper is to disseminate this information in a very understandable format, in the hopes that technology transfer w.r.t. logic programming will begin to take place. Both, the synthesis and the simplicity of this presentation of the stable model languages is absent from the literature.

II. Introduction

The goal of an automated reasoning system is to perform inferences (that is, arrive at new information) that currently elude traditional software approaches, and that seem to exhibit some sort of intelligence. Typically we approach such an endeavor in an application specific manner. That is, in trying to solve a specific problem, we limit ourselves to the domain of that specific problem. The idea of a single reasoning agent being able to reason in all domains seems impossible. Even laying aside intractable problems (that is, those problems which are mathematically provable to be impossible w.r.t NP-completeness), the thought of having a semi-omniscient reasoning agent seems far-fetched. Yet, in the ideal, this is exactly the kind of agent we desire.

Our quest is to produce a machine much like the computer "HAL" in the movies "2001: A Space Odyssey", and "2010: The Year We Make Contact". The foundation upon which such a machine can be built is logic. Significant areas of reasoning based upon logic include knowledge representation, nonmonotonic reasoning (that is, jumping to reasonable conclusions), common sense reasoning (that is, systems that are not so brittle), and deductive databases (or intelligent databases.) In short, we face many problems, including: problems with computation time (that is, fast enough computers), problems with computation space (massive amounts of memory and massive amounts of storage), identifying the corpus of knowledge that such a system must posses, and identifying proper approaches to reasoning. Another way to express this last point is to say we desire systems that can reason in semantically correct ways.

Logic programming, and in particular the semantics of logic programs, is concerned with this very issue (of reasoning in correct ways). It concerns itself with the matter of "how do we reason about such a problem"? There are competing approaches to semantics within the logic programming community. Yet, by far, the most popular and most thoroughly researched semantics is the stable model semantics.

In this paper, an overview of five logic programming languages will be given. These languages belong to a family of languages that we herein refer to as the stable model languages. These five languages are not competing languages. Rather, these languages form a strict hierarchy of expressiveness and complexity. Each level of this hierarchy is more expressive than the previous level, each level completely subsumes the expressiveness of the previous level, and each level has a greater price to pay for such expressiveness in the form of increased computational complexity. For the sake of brevity and simplicity, we will avoid a plethora of definitions, technicalities, and hair-splitting issues, all of which are the subject of the author's past work, work in progress, and planned work.

III. The Stable Model Languages

Overview

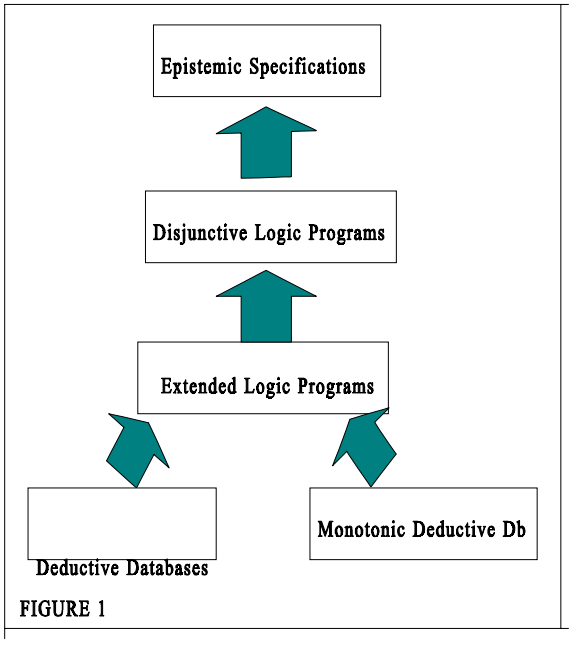

FIGURE 1

Figure 1 presents the hierarchy of stable model languages. The topmost part of the figure represents the highest, most expressive, and most complex level of the hierarchy. Conversely, the lowest part of the figure represents the lowest, least expressive, and least complex level of the hierarchy. Chronologically, the languages were developed from the bottom of the hierarchy up. Each level of the hierarchy completely subsumes the lower level. That which can be expressed at the lowest level, can be expressed in each of the higher levels, etc. Rule-based systems and Prolog slightly blur the boundaries, and perhaps belong to the class of programs at the deductive database level. (Some hair-splitting issues this paper avoids begin to arise here.)

Deductive Databases

The first and simplest semantics that we will discuss are deductive databases (Gelfond 92). A *deductive database* is a set of rules of the form:

$$A_0 \leftarrow A_1, ..., A_n$$

where A_i are ground atoms. A_0 is called the head of the rule, and $A_1, ..., A_n$ is called the body of the rule. $n \geq 0$ The A_i in the body of the rule are treated as a conjunction. A simple example of a deductive database would be the following:

$$smart\_student(X) \leftarrow merit\_scholar(X), currently\_enrolled(X)$$

which states that X is a smart student if X is a merit scholar, and if X is currently enrolled.

The semantics of deductive databases is demonstrated by figure 2. Those formula which are entailed by the program are true, and those which are not entailed by the program are false. Thus, deductive databases employ the *closed world assumption* (*CWA*). With respect to figure 2, the circle labeled true represents those facts that are explicitly listed in the database, or those additional facts which can be inferred from the database via the rules. Everything else is considered false.

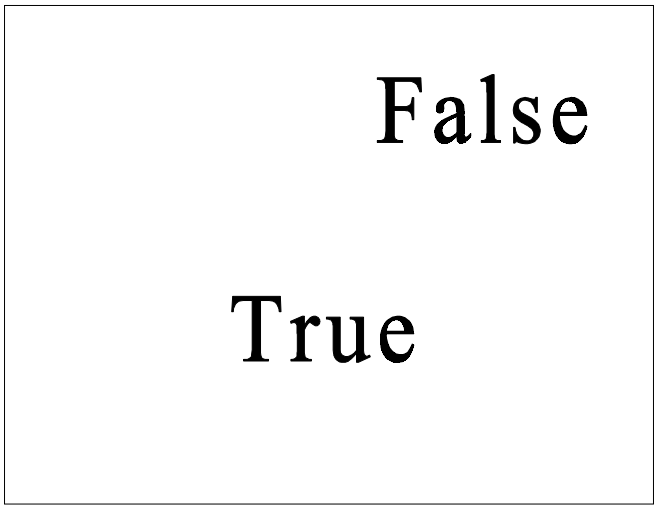

Figure 2

CWA is appropriate in applications for which it is appropriate to assume that the reasoner has complete knowledge. (Actually, there may be missing information. However, all the instances of a relation must be present. Appealing to a comparison with database technology, all the tuples must be present, even if the columns are not fully defined.) Such an application would be an airline schedule. If we ask the reasoner if there is a 3:00 flight from Dallas to Chicago on Friday, the answer is easily determined. If there exists a record in the flight table for such a flight (equivalently, if the database entails such a formula), then the answer is yes (equivalently, true.) Otherwise, it is false that such a flight exists.

Monotonic Deductive Databases

In many applications, it is a very reasonable to assume *CWA*. However, for many areas of reasoning, use of *CWA* is quite limiting, and in fact may even produce incorrect results. That is, it is too harsh to say that something is false simply because we do not know it to be true. Imagine how one would feel if a relative was on an airplane that crashed. Upon asking if the relative was alive, the system responded with "no" simply because no information existed.

A monotonic deductive database (Gelfond 92) solves this problem by removing CWA, and by allowing a new operator, ¬, to represent strong negation (Gelfond 92) (Gelfond, Lifschitz 91). Note that *CWA* is a "problem" only if it is inappropriate for the application at hand.

Rules in a monotonic deductive database are of the same form as deductive databases, except that the A_i are ground literals (Gelfond 92). This means that each A_i may or may not be preceded by the ¬ operator. A formula (precisely, an atom) preceded by the ¬ operator means that the fact is explicitly false. That is, a formula such as

$$\neg smart\_student(john)$$

means that it is absolutely false that John is a smart student. A more elaborate example of its use is the following:

$$\neg preferred\_vendor(X) \leftarrow vendor(X), past\_due\_account(X)$$

which means that X is not a preferred vendor if X is a vendor, and X is past due on its account.

The semantics of monotonic deductive databases is illustrated by Figure 3. There are two crucial, noteworthy items of interest here. First, what was previously viewed as false (i.e., if it could not be proved, it was concluded to be false) has been partitioned into that which is provably false, and that which is unknown[1]. Another way of looking at this is that that which was unknown before was concluded to be false. That which is unknown now is genuinely recognized as unknown.

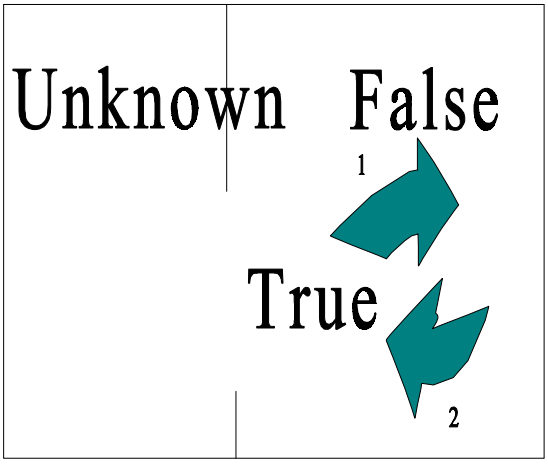

Figure 3

The second crucial point is that at this level we can generate additional inferences. We have more bases upon which to make inferences. We can use true information to prove something to be false (as illustrated by the arc labeled "1".) We can use false information to prove something to be true (as illustrated by the arc labeled "2".) We can use any combination of true and false information to infer something true (or respectively, false).

[1] Note that with this figure, and with all remaining figures, it would have been more correct to have an elipse for true, and another elipse for false. That which is false, is isomorphic with that which is true. For simplicity of understanding, these diagrams will emphasize that which is true. All statements and diagrams about that which is true apply equally to that which is false.

Extended Logic Programs

Monotonic deductive databases acknowledge that there is a distinction between that which is unknown, versus that which is provably false. Far greater expressive power is gained by being able to reason about this distinction. That is, the power to reason is greatly enhanced by being able to explicitly reason about that which is unknown. Extended logic programs (Gelfond, Lifschitz 91) allow us to do this. A new connective *not* is introduced, which intuitively means that a literal is not believed. For example, *not* $A(a)$ means that $A(a)$ is not believed (equivalently, $A(a)$ cannot be proved). On the other hand, *not* $\neg A(a)$ means that $\neg A(a)$ is not believed. The *not* operator is also called negation-as-failure. An example of this usage would be the following:

smart_student(X) ← *merit_scholar*(X),
currently_enrolled(X),
not on_academic_probation(X)

which states that X is a smart student if X is a merit scholar, X is currently enrolled, and X is not on academic probation.

Note that saying that it cannot be proved that X is on academic probation is much weaker (and more useful since much of life is unknown) than stating that it is definitively false that X is on academic probation. That is,

not on_academic_probation(X)

is a weaker statement than

¬*on_academic_probation*(X).

The semantics of extended logic programs are illustrated by figures 4 and 5. These semantics incorporate the semantics of the previous level of our hierarchy. That is, these semantics also distinguish between that which is unknown and that which is false, and these semantics allow the additional inferences possible by reasoning about both, true and provably false information.

Additionally, the semantics of extended logic programs allow us to reason explicitly about the fact that something is unknown (as illustrated by the *not on_academic_probation*(X) above.) The arcs labeled by "1" and "2" illustrate that new inferences can be made on the basis of unknown information. Arc "1" illustrates that true information can be inferred from unknown information. Arc "2" illustrates that provably false information can be inferred from unknown information. (Of course, positive inferences, and negative inferences can be made on the basis of any combination of positive information, negative information, or unknown information.)

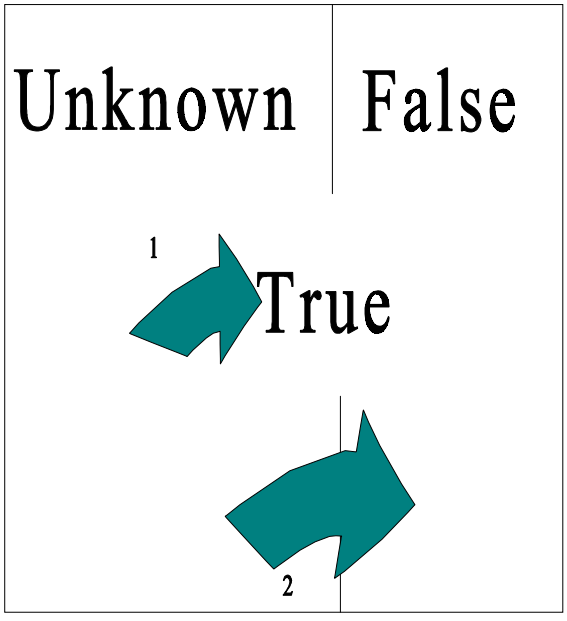

Figure 4

Unfortunately, figure 4 represents only part of the story. Negation-as-failure (that is, use of the *not* operator) creates the possibility of having multiple belief sets. (That is, multiple models, or multiple ways of viewing the world.). For space considerations, let us refer to a future figure: consider figure 5 without the arcs. In figure 5, there are multiple (two) ellipses that are labeled "yes". Each ellipse represents a model of the world. It is only the intersection of the ellipses that represent that which is true with respect to the program. (Similarly, but not shown by the figure, there are multiple areas that should be labeled "false". It is only the intersection of all those areas which represent that which is false with respect to the program.) All else is unknown. Multiple models represent some computational problems. For instance, circumstances could be such that a program may go into an infinite loop when computing these models. Therefore, a substantial amount of research is invested in identifying those classes of programs which are "safe". Much work has been done to identify which programs have unique models. Much work has also been done to identify other classes of programs which pose no problem, even though they do have multiple models.

Disjunctive logic programs

Disjunction is the facility that allows us to represent the fact that we know that at least one of two possibilities is true, but that we do not know which one of the two is true. To represent disjunction, a new connective *or* is introduced (Gelfond, Lifschitz 91). This operator is called *epistemic disjunction*, and appears only in the heads of rules. A formula of the form

$$A \text{ or } B \leftarrow$$

is interpreted to mean "*A* is believed to be true or *B* is believed to be true, but both are not true." A disjunctive deductive database is a collection of rules of the form:

$$L_0 \text{ or } \ldots \text{ or } L_k \leftarrow L_{k+1}, \ldots, L_m, \text{ not } L_{m+1}, \ldots, \text{ not } L_n$$

where L_i are literals (as before.)

Again, figure 5 without the arcs is representative of the semantics of disjunctive logic programs. While it appears that the semantics of disjunctive logic programs are the same as the semantics of extended logic programs, (Eiter, et al. 97) proves that there are problems that can be represented by disjunctive logic programs that cannot be represented by extended logic programs.

Epistemic Specifications

There is a major intuitive problem with the posture of requiring that all models must agree on the truth or falsity of a formula in order for that formula to be true or false. (Such is the case with extended logic programs, and with disjunctive logic programs.) For instance, there is a fundamental difference between claiming something is unknown because we have no earthly idea about it, and claiming something is unknown because the multiple models cannot agree on it. In the later case, we do have some idea about the formula in question, it is just that we do not have a unanimous consent about the formula.

Epistemic specifications (Gelfond 92) introduces modal operators to allow us to introspect among our belief sets. In particular, the modal operator *M* signifies that something *may* be true (that is, it is true in at least one belief set). The modal operator *K* signifies that something is *known* to be true (that is, it is true in all belief sets.) As illustrated in Figure 5 the arc labeled "1" indicates additional inferences that may be made by ascertaining that something is true in all belief sets. The arc labeled "2" indicates additional inferences that may be made by

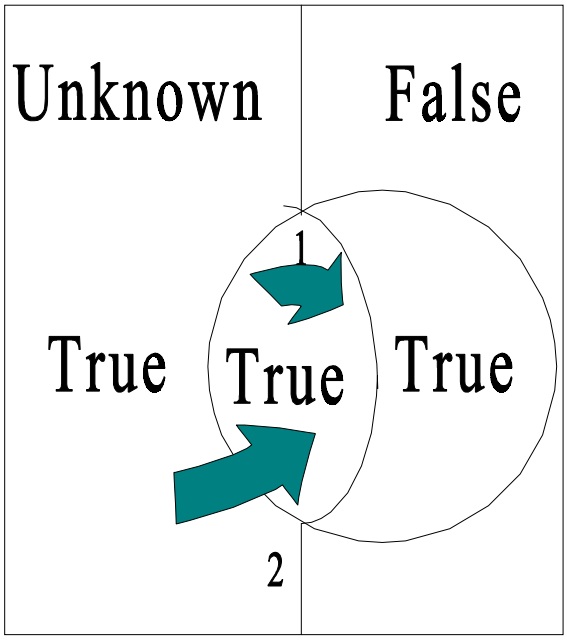

Figure 5

ascertaining that something is true in at least one belief set. Further, positive information or negative information can be inferred on the basis of any combination of: positive information, negative information, unknown information, or recursively any of this information which is true in any or all models.

An example of the usefulness of introspecting among belief sets would be the following:

broken_arm(*john*) or *spained_arm*(*john*) ←
order(*plaster*) ← *M broken_arm*(*X*)

This example states that John has either a broken arm, or a sprained arm (but not both). Wanting to be prepared for any situation which may deplete our inventory, we wish to order medical supplies (i.e., plaster) to meet potential needs. This example has two belief sets: one in which *broken_arm*(*john*) is true, and the other in which *spained_arm*(*john*) is true. The 2nd rule states that if there is any belief set in which an individual has a broken arm, we want to order some plaster. This is achieved by the *M* modal operator. Therefore, there are two belief sets for this program: {*broken_arm*(*john*), *order*(*plaster*)} and

{*spained_arm*(*john*), *order*(*plaster*)}. Since *order*(*plaster*) appears in both belief sets, plaster will indeed be ordered.

A truer picture of the semantics of epistemic specifications would be a 3-dimensional view of figure 5. Figure 5, which is one plane would be referred to as one worldview. Each worldview can consist of or more belief sets. There may be multiple worldviews. We can conceptually think of this as several parallel planes. It is the intersection of all worldviews that determines that which is false and that which is true with respect to the program. (Imagine the planes as being collapsed into one plane so that there is an intersection of all the worldviews.) All else is unknown. However, now with the ability to introspect among belief sets, the unknown is truly unknown.

Concluding Remarks

Each of the 5 semantics presented may be the appropriate solution for particular problems. However, as a practical matter, the language of extended logic programs is the language of choice for most applications. Extended logic programs represent the best compromise between expressive power and performance. Further, the languages lower in the hierarchy than extended logic programs (deductive databases and monotonic deductive databases) can be implemented as extended logic programs. Disjunctive logic programs, and epistemic specifications provide incredible reasoning power. However, these languages introduce computational complexities that make their implementation for large applications impractical at this time.

Monotonic deductive databases can be implemented as extended logic programs by disallowing the operator *not*. Without *not*, extended logic programs have the exact same form as monotonic deductive databases. The semantics of entailment would be exactly the same as well.

It takes a bit more work to implement deductive databases as extended logic programs. First, let us point out that *CWA*, which is inherent in deductive databases, can be selectively implemented in extended logic programs. Consider a human resources database that implements the *employee* relation with rules of the form:

employee(*john, accounting, 40000*)

which states that *john* is an employee who works in the *accounting* department, earning a salary of *$40,000.* Suppose we ask the query (or similarly, try to prove the goal)

employee(*mary, X, Y*)

which asks whether or not *mary* is an employee. Suppose that such a fact is not provable in our database. Interpreted as a deductive database, the answer would be *no* (or *false*). Interpreted as an extended logic program, the result would be *unknown*. We could achieve the desired result in an extended logic program by applying *CWA* to the *employee* relations. This would be done with the following rule:

$$\neg employee(X, Y, Z) \leftarrow not\ employee(X, Y, Z)$$

This rule means that if it is not believed that *X* is an employee, then it is definitively false (by way of default reasoning) that *X* is an employee. To implement a deductive database as an extended logic program, for each predicate, we would have to create a rule similar to the one above.

As an indication of the "state of the art" in research, the logic programming research community is currently enamored with disjunctive logic programs. Epistemic specifications possess far superior reasoning abilities. However, implementations of these languages are not yet efficient enough to be seriously considered for applications development at this time. Further, the deep properties of this language are not yet well understood.

References

(Apt, Bol 94) Apt, Krzysztof R., and Roland N. Bol: Logic Programming and Negation: A Survey, *Journal of Logic Programming*, vol 19/20 May/July 1994.

(Baral, Gelfond 94) Baral, Chitta, and Michael Gelfond: Logic Programming and Knowledge Representation, *Journal of Logic Programming*, vol 19/20 May/July 1994.

(Deransart, Maluszynski 93) Deransart, Pierre, and Jan Maluszynski: *A Grammatical View of Logic Programming*, Cambridge Massachusetts: The MIT Press

(Eiter et al. 97) Eiter, Thomas, Georg Gottlob, and Heikki Mannila: Disjunctive Datalog, *ACM Transactions*

on Database Systems, vol 22, no. 3, September 1997)

(Gelfond 92) Gelfond, Michael: Logic Programming and Reasoning with Incomplete Information (to appear in *The Annals of Mathematics and Artificial Intelligence*, 1994)

(Gelfond, Lifschitz 88) Gelfond, Michael and Vladimir Lifschitz: The Stable Model Semantics for Logic Programming, *5th Intl Conference on Logic Programming* 1988

(Gelfond, Lifschitz 90) Gelfond, Michael, and Vladimir Lifschitz: Logic Programs with Classical Negation. In D. Warren and Peter Szeredi, editors, *Logic Programming: Proceedings or the 7th Int'l Conf*, 1990.

(Gelfond, Lifschitz 91) Gelfond, Michael, and Vladimir Lifschitz: Classical Negation in Logic Programs and Disjunctive Databases, *New Generation Computing*, No. 9 1991

(Genesereth, Nilsson 87) Genesereth, Michael R., and Nils J. Nilsson: *Logical Foundations of Artificial Intelligence*, Los Altos, Ca.: Morgan Kaufman

(Lifschitz, 89) Lifschitz, Vladimir: Logical Foundations of Deductive Databases, *Information Processing* 89, North-Holland

(Lloyd 87) Lloyd, J.W.: *Foundations of Logic Programming*, Berlin, Germany: Springer-Verlag

(Reeves, Clarke 90) Reeves, Steve, and Michael Clarke: *Logic for Computer Science*, Workingham, England: Addison-Wesley Publishing Co.