Set Semantics and Operations Based Upon Answer Set Semantics

James D. Jones

Computer Science
College of Information Science and Systems Engineering
University of Arkansas at Little Rock
2801 S. University Av.
Little Rock, AR. 72204-1099
james.d.jones@acm.org

I. Abstract

We propose to extend the language of *extended logic programs* by providing a declarative semantics for the use of *sets* as terms. Such an extension is intuitively natural, since humans frequently reason about collections of items. Our approach to defining a declarative semantics for sets differs from other approaches in that other approaches are very limited in their use of negation, and our approach is model-theoretic in a context allowing multiple models. Further, our approach is based on the answer set semantics, providing a well understood foundation. The language defined here can represent a wide range of common sense problems. As a side effect, an auxiliary contribution of this work is that a subset of this work provides a declarative semantics for how the *setof* symbol and how sets as terms should be and can be implemented in Prolog..

II. Motivation and Preview

In this paper, we extend the language of *extended logic programs* (Baral, Gelfond 94) by defining a declarative semantics for sets as terms. This is achieved by the addition of two built-in predicates, *setof* and *r*. Also added are the punctuation symbols **{**, **}**, **[**, and **]**. While there are other approaches for defining semantics for sets (Beeri, et. al. 91, Dovier et al. 96, Gervet 94, Jayaraman, Plaisted 89, Kuper 90), it is believed that our semantics is a) simpler, and b) more expressive (as a consequence of being defined in a rich family of languages). The greater expressiveness results from a) allowing arbitrary set terms (for example, any level of nesting of sets and lists), and b) allowing programs that have multiple belief sets. (Although, the proof of these claims is beyond the scope of this paper.)

The primary motivation for these extensions is that many real-life queries (or processing requests) are of the form "give me the set of all ...." For example, one may ask, "give me the set of all employees which have not received a raise in the past two years." Other types of queries involving sets are queries involving *ranking*. For example, one may ask "who are the top 10 active baseball players," or "what rank is John Doe." One must know the set of all baseball players, and the evaluation criteria, in order to answer such questions. Of course, one could "hard code" such an answer with something like *rank("john doe"*, 3). However, such an approach is not *general purpose*, and is fraught with problems.

As will be defined more precisely later, to answer such set related questions, we will define the predicate *setof*. For readers familiar with Prolog, the syntax will be very similar to the syntax of the built-in predicate of the same name. That is, it will be of the form:

$$setof(X, F, S)$$

where the first argument will contain the variables for which we wish to "collect" data; the second argument will contain the formula(e), or criteria, which define the data we want to collect; and the final argument will be the set into which the results will be placed. Again, for readers familiar with Prolog, the semantics of *setof* in our language has similarities to the procedural meaning of the built-in Prolog predicate *setof*, where the unbound variables are universally quantified.

Our semantics is based on entailment. This fact provides the fundamental distinction between our approach and other approaches. The only elements which can appear in such a set are those terms appearing in ground instances of the formula(e) contained in the *setof* predicate which are entailed by the belief set(s).

III. Formalization of the Language

A. Syntax

The language that we are considering will consist of predicate constants, function constants, object constants, object variables, the connectives ¬, ***not***, and ←, and the symbols **,** (the comma), **{**, **}**, **[**, and **]**. *setof* and *r* are special predicate constants defined by the language.

Definition: A *term* is defined inductively as follows:

1. an object constant or an object variable is a term.
2. if f is an n-ary function symbol and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.
3. if $t_1, ..., t_n$ $n \geq 0$ are terms, then $\{t_1, ..., t_n\}$ is a term. (Terms of this form are called *set terms*, and have the meaning normally associated with sets.)
4. if $t_1, ..., t_n$ $n \geq 0$ are terms, then $[t_1, ..., t_n]$ is a term. (Terms of this form are *lists*.)

Note that item 3 of the definition of a term is the major contribution of this definition. Note also that this definition allows for arbitrary terms: terms with any level of nesting of sets, and sets whose elements themselves are arbitrary (formed from any traditional term, set or list.)

Definition: An *atom* is of the form $P(t_1, ..., t_n)$, where P is an n-ary predicate constant, the t_i $0 \leq i \leq n$ are terms.

Definition: A *literal* is an atom or its strong negation (i.e., $P(t_1, ..., t_n)$ or $\neg P(t_1, ..., t_n)$).

Definition: An *extended literal* is a literal preceded by *not*.

Definition: A *formula* is defined inductively as follows:

1. a literal is a formula
2. an extended literal is a formula
3. if F and G are formulae, then F,G is a formula, read as a conjunction, F and G
4. if $F(\bar{X})$ is a formula, then $setof(\theta(\bar{X}), F(\bar{X}), Y)$ is a formula
   where
   $\bar{X}$ is an n-tuple of variables appearing in $F(\bar{X})$, $n \geq 1$
   $\theta(\bar{X})$ and Y are arbitrary terms
5. if $F(\bar{X})$ is a formula, then $\neg setof(\theta(\bar{X}), F(\bar{X}), Y)$ is a formula
   where
   $\bar{X}$ is an n-tuple of variables appearing in $F(\bar{X})$, $n \geq 1$
   $\theta(\bar{X})$, Y are arbitrary terms ☐

Note that items 4 and 5 of the definition of a formula are the primary contribution of this definition. For brevity, we accept the common definitions for ground atoms, etc.

Definition: An *extended logic program with setof* consists of rules of the form:

$L \leftarrow F_1, ..., F_n$

where

L is a literal
F_i are formulae
$n \geq 0$ ☐

B. Semantics

1. Ground Rules

In the following, we consider the Herbrand interpretation of symbols. Furthermore, we consider only ground rules. Rules containing variables, except for $\bar{X}$ in $setof(\theta(\bar{X}), F(\bar{X}), Y)$ are meant to represent schemas for ground instances of rules. The Y in the *setof* formula of such rules are replaced only by finite, ground, set terms.

2. Entailment With Respect to a Belief Set

Definition: Given an extended logic program with *setof* Π, *Lit* is the set of ground literals in the language of Π.

Definition: The *definition of a literal l* is the collection of all ground instances of l which are heads of rules.

Let Π be an *extended logic program with setof.* Let $S \subseteq Lit$ be a set of literals such that for every literal $l \in S$ appearing in a formula in Π involving *setof*, the definition of l is finite.

Definition: *Entailment w.r.t. S.* Let Q be some ground formula.

case 1: Q is of the form $setof(\theta(\bar{X}), F(\bar{X}), Y)$.
$S \models Q$ iff Y is $\{\theta(\bar{t})$: $S \models F(\bar{t})$ where $\theta(\bar{t})$ is an arbitrary term formed from the tuple of ground terms which are among the ground terms appearing in $F\}$.

case 2: Q is of the form $\neg setof(\theta(\bar{X}), F(\bar{X}), Y)$.
$S \models Q$ iff Y is not $\{\theta(\bar{t})$: $S \models F(\bar{t})$ where $\theta(\bar{t})$ is an arbitrary term formed from the tuple $\bar{t}$ of ground terms which are among the ground terms appearing in $F\}$.

case 3: Q is a literal. $S \models Q$ iff $Q \in S$.

case 4: Q is an extended literal, that is, of the form *not* P. $S \models Q$ iff $P \notin S$. ☐

Note that the difficulty in defining entailment of *setof* is the potential self-referential nature of *setof*. An approach to semantics based on fixed points could lead to an endless cycle. At the termination of each loop through this cycle there could exist a different set. Entailment based on this different set could yield results that perpetuate the cycle. Our semantics avoids this difficulty by first fixing a set S, and then defining entailment with respect to this fixed set. This same difficulty, and this same solution exists for defining a semantics for negation as failure.

Definition: S *satisfies* a rule iff for every formula F in the body of the rule, if $S \vDash F$, then L, the head of the rule, is an element of S. That is, $L \in S$.

Definition: S is a *belief set* of Π iff S is a minimal set satisfying all the rules of Π.

Example 1
Let Π be the following program:

$$Pa \leftarrow$$
$$Pb \leftarrow$$
$$R(Y) \leftarrow setof(X, P(X), Y)$$

Replacing the last rule with all the ground instances of the schema, our program is the following:

(1) $Pa \leftarrow$
(2) $Pb \leftarrow$
(3) $R(\{\}) \leftarrow setof(X, P(X), \{\})$
(4) $R(\{a\}) \leftarrow setof(X, P(X), \{a\})$
(5) $R(\{b\}) \leftarrow setof(X, P(X), \{b\})$
(6) $R(\{a, b\}) \leftarrow setof(X, P(X), \{a, b\})$

Let $S = \{Pa, Pb, R(\{a, b\})\}$. Rules 3 thru 5 are satisfied by S, since neither the bodies, nor their heads are entailed by S. The body of rule 6 is entailed by S, therefore $R(\{a, b\})$ should be an element of S. Rules 1 and 2 are trivially entailed by S. Since S is a minimal set satisfying the rules of this program, S is a belief set of this program. There are no other belief sets of this program. □

3. *Entailment With Respect to a Program*

Negation-as-failure creates the possibility that a program may have a unique belief set, may have no belief set, or may have more than one belief set. The semantics defined so far in this paper does not accommodate multiple belief sets. We will now extend our semantics to accommodate multiple belief sets. This is the key point at which we believe our semantics to be superior to other attempts to define a semantics for sets. We believe that there are also other reasons that our semantics are superior, but the ability to reason in the presence of multiple belief sets is clearly the most significant. It is because our semantics is based upon entailment that our semantics possesses this ability.

Definition: *p_set_of* is a special predicate constant belonging only to the query language. It is of the same syntactical form as *setof*.

p_set_of is introduced only as a convenience for the reader, to avoid confusion between entailment w.r.t. a belief set, and entailment w.r.t. a program. This distinction is important only when the program has multiple belief sets. For all intents and purposes, *setof* could be used throughout. All the previous definitions regarding grounding, formulas, etc. are correspondingly modified to use *p_set_of* instead of *setof* in the appropriate context defined as follows. Formulae involving *setof* appear only in the bodies of rules. Formulae involving *p_set_of* appear only as queries to programs.

Definition: *Entailment w.r.t.* Π. Let Q be some ground formula. Let $\alpha(\Pi) = \{S\colon S$ is a belief set of $\Pi\}$.

case 1: Q is of the form $p\_set\_of(\theta(\overline{X}), F(\overline{X}), Y)$. $\Pi \vDash Q$ iff Y is $\{\theta(\overline{t})\colon\ S \vDash F(\overline{t})$ for all $S \in \alpha(\Pi)$, where $\theta(\overline{t})$ is an arbitrary term formed from the tuple $\overline{t}$ of ground terms appearing in $F\}$.
case 2: Q is of the form $\neg p\_set\_of(\theta(\overline{X}), F(\overline{X}), Y)$. $\Pi \vDash Q$ iff Y is not $\{\theta(\overline{t})\colon\ S \vDash F(\overline{t})$ for all $S \in \alpha(\Pi)$, where $\theta(\overline{t})$ is an arbitrary term formed from the tuple $\overline{t}$ of ground terms appearing in $F\}$.
case 3: Q is a literal. $\Pi \vDash Q$ iff $Q \in S$ for all $S \in \alpha(\Pi)$.
case 4: Q is an extended literal, that is, of the form *not* P, where P is a ground literal. $\Pi \vDash Q$ iff $P \notin S$ for some $S \in \alpha(\Pi)$. □

Definition: *answers to queries*. Let Q be some sequence of ground formulae, $q \in Q$ be a ground formula. For any such Q posed as a query to Π, Π answers *yes* iff $\Pi \vDash q$ for all $q \in Q$, *no* iff $\Pi \vDash \neg q$ for some $q \in Q$, and *unknown* otherwise. (That is, $\Pi \vDash Q$ iff $\Pi \vDash q$ for all $q \in Q$; $\Pi \vDash \neg Q$ iff $\Pi \vDash \neg q$ for some $q \in Q$; otherwise $\Pi \nvDash Q$ and $\Pi \nvDash \neg Q$.)

Example 2
This example demonstrates entailment with respect to a program, and demonstrates the difference between entailment with respect to a belief set and entailment with respect to a program. More importantly, this example illustrates that caution must be used when reasoning about sets which are entailed by a program. It may not be clear what the user is actually reasoning about when reasoning with sets.

Assume we have the following program. Assume also that there exists a defined predicate *number* which determines the cardinality of a set term.

$$attack(fred, me) \leftarrow$$
$$attack(john, me) \leftarrow not\ attack(bill, me)$$
$$attack(bill, me) \leftarrow not\ attack(john, me)$$
$$setof\_attackers(Y) \leftarrow setof(X, attack(X, me), Y)$$

This program has two answer sets, S_1 and S_2 defined as follows.

$S_1 = \{attack(fred, me), attack(john, me), setof\_attackers(\{fred, john\})\}$
$S_2 = \{attack(fred, me), attack(bill, me), setof\_attackers(\{fred, bill\})\}$

Note that belief set S_1 entails

$$setof(X, attack(X, me), \{fred, john\})$$

and that belief set S_2 entails

$$setof(X, attack(X, me), \{fred, bill\}).$$

Suppose that the user of this program has the following informal rule: "if the number of attackers is 1, then fight, otherwise run." Suppose we ask the query

$$\leftarrow p\_set\_of(X, attack(X, me), Y)$$

The program entails

$$p\_set\_of(X, attack(X, me), \{fred\}).$$

The user of such a program would be led to believe that there was only one attacker, hence, "fight." Yet, it is clear that in each of the belief sets, the number of attackers is 2, hence, the user should have "run." This is not a problem with the semantics. Rather, this example demonstrates that one must make careful use of such a system. The proper results would be obtained if we added the following rules to the program. (We assume the standard interpretation for integers.)

$$fight \leftarrow set\_of\_attackers(S), number(S, N), N \leq 1$$
$$run \leftarrow set\_of\_attackers(S), number(S, N), N \geq 1$$

The two answer sets for this program are:

$S_3 = \{run, attack(fred, me), attack(john, me), setof\_attackers(\{fred, john\})\}$
$S_4 = \{run, attack(fred, me), attack(bill, me), setof\_attackers(\{fred, bill\})\}$

Hence, the program properly entails *run*. What sets this example apart is that we are not concerned about the set entailed by the program. Rather, we are concerned about characteristics (in this case, cardinality) of the sets entailed by all the belief sets. Further, this example illustrates that it is vitally important to consider entailment with respect to a belief set and entailment with respect to a program. A set entailed by a belief set may implicate the contents of that belief set, which in turn implicates what is entailed by a program. We have a concomitant entailment. □

C. Set Operations

Definition: $r(X, S_1, S_2)$ is an atom. This atom is true when $X \in S_1$, and set S_2 is the set obtained by removing element X from set S_1.

All the basic set operations can be defined in the language itself, using the built-in predicate $r(X, S_1, S_2)$. Following are definitions for *union*, *intersection*, *difference*, *cardinality*, *member*, *equal*, and *subset*.

- *member*(X, S)
  X is a member of set S

$$member(X, S) \leftarrow r(X, S, S_1)$$

- *union*(S_1, S_2, S_3)
  the union of sets S_1 and S_2 is set S_3

$$union(\{\ \}, S, S).$$
$$union(S_1, S_2, S_3) \leftarrow r(X, S_1, S_{1a}), r(X, S_{2a}, S_2), union(S_{1a}, S_{2a}, S_3).$$

The basic idea behind *union* is to recursively : 1) define a set S_{1a} which is composed of all but one of the elements of set S_1, 2) define a set S_{2a} which is composed of all of the elements of set S_2 and the element in the difference set of S_1 - S_{1a}, and 3) union the two new sets, S_{1a} and S_{2a}.

- *intersection*(S_1, S_2, S_3)
  the intersection of sets S_1 and S_2 is set S_3

$$
\begin{aligned}
&intersection(S, S, S).\\
&intersection(\{\}, S, \{\}).\\
&intersection(S_1, S_2, S_3) \leftarrow\ r(X, S_1, S_{1a}),\\
&\qquad r(X, S_2, S_{2a}),\\
&\qquad r(X, S_3, S_{3a}),\\
&\qquad intersection(S_{1a}, S_{2a}, S_{3a}) .\\
&intersection(S_1, S_2, \{\}) \leftarrow\ r(X, S_1, S_{1a}),\\
&\qquad not\ member(X, S_2),\\
&\qquad intersection(S_{1a}, S_2, \{\}) .
\end{aligned}
$$

The meaning of the first two rules is straightforward: a set intersects itself, and the intersection of a set and the empty set is the empty set. The basic idea behind the third rule is to define three new sets by removing the same element from the original sets, and determine the truth of *intersection* involving these new sets. The basic idea behind the last rule is that two sets are disjoint if no element of the first set is also an element of the second set.

- *difference*(S_1, S_2, S_3)
  the set difference between S_1 and S_2 is S_3

$$
\begin{aligned}
&difference(S, \{\}, S).\\
&difference(S_1, S_2, S_3) \leftarrow r(X, S_1, S_{1a}),\\
&\qquad r(X, S_2, S_{2a}),\\
&\qquad difference(S_{1a}, S_{2a}, S_3)\\
&difference(S_1, S_2, S_3) \leftarrow r(X, S_2, S_{2a}),\\
&\qquad not\ member(X, S_1),\\
&\qquad difference(S_1, S_{2a}, S_3)
\end{aligned}
$$

The meaning of the first rule is straightforward. The second rule is utilized in those cases where an element occurs in each of the sets under consideration (i.e., S_1 and S_2). The last rule is utilized in those instances where an element occurs in the second set, and not in the first set. We want to eliminate such elements from consideration.

- *cardinality*(S, N)
  the cardinality of set S is N

$$
\begin{aligned}
&cardinality(\{\}, 0).\\
&cardinality(S, N) \leftarrow r(X, S, S_1),\\
&\qquad cardinality(S_1, N_1),\\
&\qquad N\ is\ N_1 + 1.
\end{aligned}
$$

The meaning of the first rule is straightforward: the empty set has no elements. The last rule states that the cardinality of a set is one greater than the cardinality of a set composed from this same set with one element removed.

- *equal*(S_1, S_2)
  set S_1 is equal to set S_2

$$
\begin{aligned}
&equal(\{\}, \{\}).\\
&equal(S_1, S_2) :-\\
&\qquad r(S, S_1, S_{1a}),\\
&\qquad r(S, S_2, S_{2a}),\\
&\qquad equal(S_{1a}, S_{2a}).
\end{aligned}
$$

Two sets are equal if the same element can be removed from both sets, and the resulting sets are equal.

- *subset*(S_1, S_2)
  set S_1 is a subset of set S_2

$$
\begin{aligned}
&subset(\{\}, S).\\
&subset(S_1, S_2) \leftarrow\ r(X, S_1, S_{1a}),\\
&\qquad r(X, S_2, S_{2a}),\\
&\qquad subset(S_{1a}, S_{2a}).
\end{aligned}
$$

The empty set is a subset of every set. The basic idea behind the last rule is that one set is a subset of another if we can recursively remove the same element from each, and determine that *subset* is true of the resulting sets.

IV. Final Comments

We have defined a syntax and a declarative semantics for the use of sets as terms in the context of the answer set semantics. The syntax is powerful, allowing arbitrary set terms to be represented at any level of nesting. The semantics is simple, being defined with respect to entailment. We have clearly differentiated between that which is entailed by a belief set, and that which is entailed by a program. This distinction is important only when a program has multiple belief sets.

For programs with unique belief sets, that which is entailed by a belief set coincides with that which is entailed by a program. The answer set languages are very expressive in that they can represent a wide range of common sense problems. Endowing these languages with the ability to reason about sets opens up a very large class of problems that can be properly represented and reasoned about.

It is believed that expanding this work to infinite sets is relatively straightforward. It is also believed that applying this semantics for sets to the more advanced answer set languages (epistemic disjunction, and epistemic specifications) is relatively straightforward. Future work must consider more recent work on set semantics.

Acknowledgments

I wish to thank my teacher and mentor, Michael Gelfond, for the many fruitful discussions that led to this work.

Bibliography

(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.

(Beeri, et. al. 91) Beeri, S. Naqvi, O. Shmueli, and S. Tsur: Set Constructors in a Logic Database Language, *Journal of Logic Programming*, 10(3):181-232, 1991.

(Dovier et al. 95) Dovier, Agostino, Enrico Pontelli, and Gianfranco Rossi: The *CLP* Language {log}, and the relation between Intensional Sets and Negation, New Mexico State University technical report NMSU-CSTR-9503, March 95.

(Dovier et al 96) Dovier, Agostino, Enrico Pontelli, and Gianfranco Rossi: {log}: A language for programming in logic with finite sets, *Journal of Logic Programming*, 28(1):1-44, 1996.

(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

(Gervet 94) Gervet, Carmen.: Constraint Logic Programming with Finite Set Domains, *Proceedings of International Logic Programming Symposium*, 1994

(Gervet 97) Gervet, Carmen: Interval propagation to reason about sets: Definition and implementation of a practical language, Constraints, 1(3):191-244, 1997.

(Jayaraman 92) Jayaraman, B.: Implementation of Subset-Equational Programs, *Journal of Logic Programming*, 12(4):299-324, 1992

(Jayaraman, Plaisted 89) Jayaraman, B., and D. A. Plaisted: Programming with Equations, Subsets and Relations, *Proceedings of NACLP89*, MIT Press, 1989

(Kuper 90) Kuper, Gabriel: Logic Programming with Sets, *Journal of Computer and System Science*, 1990

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