New Generation Computing, 9 (1991) 387-400
OHMSHA, LTD. and Springer-Verlag

Logic Programs with Exceptions*

Robert A. KOWALSKI and Fariba SADRI
Department of Computing,
Imperial College,
London SW7 2BZ, U. K.

Received 2 November 1990
Revised manuscript received 1 April 1991

Abstract We extend logic programming to deal with default reasoning by allowing the explicit representation of exceptions in addition to general rules. To formalise this extension, we modify the answer set semantics of Gelfond and Lifschitz, which allows both classical negation and negation as failure.

We also propose a transformation which eliminates exceptions by using negation by failure. The transformed program can be implemented by standard logic programming methods, such as SLDNF. The explicit representation of rules and exceptions has the virtue of greater naturalness of expression. The transformed program, however, is easier to implement.

Keywords: Logic Programs, Rules and Exceptions, Default Reasoning, Answer Set Semantics, e-answer Set Semantics

§1 Introduction

In this paper we extend logic programming to include explicit representation of exceptions. Thus we can represent both general rules such as

$$\text{fly}(X) \leftarrow \text{bird}(X)$$

and exceptions such as

$$\neg\text{fly}(X) \leftarrow \text{ostrich}(X)$$
$$\neg\text{fly}(X) \leftarrow \text{penguin}(X).$$

If John is both an ostrich and a bird, then we conclude that John does not fly, because the exception overrides the general rule.

* This paper is a revised version of the paper presented at the Seventh International Conference on Logic Programming, 1990, Jerusalem.

Rules and exceptions are a natural way of understanding default reasoning. They are expecially common in the formulation of legislation and in reasoning about inheritance in hierarchies. Our treatment of rules and exceptions can be regarded as an adaptation, within a logic programming framework, of Poole's approach[11] to default reasoning.

The contribution of this paper is two-fold. We describe a semantics for logic programming with rules and exceptions. We also present a transformation which eliminates the distinction between rules and exceptions by using negation as failure.

The semantics for rules and exceptions is a modification of the answer set semantics of Gelfond and Lifschitz[5] for extended logic programs (defined in Section 3.1 below). The answer set semantics is a generalisation of stable model semantics.[4] Both Poole[11] and Gelfond and Lifschitz[5] have shown close connections between their work and Reiter's default logic.[12] There is a similarly close connection between default logic and our extension of logic programming.

Applied to the example above, the transformation gives the following program.

$$\begin{aligned} \text{fly}(X) &\leftarrow \text{bird}(X), \text{ not } \neg\text{fly}(X) \\ \neg\text{fly}(X) &\leftarrow \text{ostrich}(X) \\ \neg\text{fly}(X) &\leftarrow \text{penguin}(X). \end{aligned}$$

Here, and throughout the paper, "$\neg$" denotes classical negation and "not" denotes negation by failure. The transformation generalises a similar transformation by Asirelli et al.[1] for eliminating certain kinds of integrity constraints from definite clause databases. The transformed program is equivalent to the original in the sense that the two programs have the same semantics.

As Gelfond and Lifschitz[5] show, classical negation can be eliminated from extended logic programs by means of renaming. The same renaming can be used for rules and exceptions. For example:

$$\begin{aligned} \text{fly}(X) &\leftarrow \text{bird}(X), \text{ not nofly}(X) \\ \text{nofly}(X) &\leftarrow \text{ostrich}(X) \\ \text{nofly}(X) &\leftarrow \text{penguin}(X). \end{aligned}$$

Moreover, in this example the new predicate symbol and the exceptions can be eliminated by macroprocessing (or equivalently, by unfolding or partial evaluation):

$$\text{fly}(X) \leftarrow \text{bird}(X), \text{ not ostrich}(X), \text{ not penguin}(X).$$

The resulting program is essentially equivalent to the original. It can be implemented using ordinary methods for logic programs with negation by failure, such as SLDNF[9] or abduction.[3]

To summarise, this paper presents an extension of logic programming to include explicit representation of exceptions and a transformation which eliminates exceptions by using negation by failure. The explicit representation improves naturalness of expression, whereas the use of the transformation facilitates implementation. The transformation has been implemented in Prolog.[6]

The paper is structured as follows. Section 2 gives an example of rules and exceptions from legislation. Section 3 formally describes the extension of logic programming to include our language of rules and exceptions and its semantics. Section 4 describes the transformation and proves that it preserves the semantics of the original program. Section 5 discusses related work.

This paper is a revision of an earlier report.[7] The main difference is that in the report we interpreted exceptions as integrity constraints, and we associated with each such integrity constraint a unique way of revising the rules in order to restore consistency after a violation of integrity. In this paper we incorporate rule-revision into a modified answer set semantics.

The main advantage of the earlier integrity constraint interpretation is that it did not require the need to define a new semantics for rules and exceptions. The main advantage of the modified answer set semantics is that it greatly simplifies the proof that the transformation preserves the meaning of the original program.

Throughout the paper we use the following notational convention. Predicate, function and constant symbols start in the lower case, and variables start in the upper case.

§2 An Example from Legislation

Elsewhere[13] we have argued, partly on the basis of our formalisation of the 1981 British Nationality Act, that legislation written in natural language approximates the form of a logic program. A closer reading, however, discloses many provisions which have a negative form. Such provisions typically express exceptions.

Section 12, subsection (1), for example, expresses a general rule

> "If any British citizen of full age and capacity makes in the prescribed manner a declaration of renunciation of British citizenship, then subject to subsections (3) and (4), the Secretary of State shall cause the declaration to be registered."

Section 12, subsection (3) expresses an exception to the rule

> "A declaration made by a person in pursuance of this section shall not be registered unless the Secretary of State is satisfied that the person who made it will after the registration have or acquire some citizenship or nationality other than British citizenship; ..."

Formally, subsections (1) and (3) can be expressed in the form

$$P \leftarrow Q$$
$$\neg P \leftarrow \text{not } R.$$

Here the negation implicit in the word "unless" has been interpreted as negation by failure, "not", rather than classical negation "$\neg$".

Using the transformation defined later in this paper, subsections (1) and (3) can be transformed into rules of the form

$$P \leftarrow Q, \text{ not } \neg P$$
$$\neg P \leftarrow \text{not } R$$

which, in turn, can be simplified to

$$P \leftarrow Q, R$$

which expresses that

> **If** any British citizen of full age and capacity makes in the prescribed manner a declaration of renunciation of British citizenship, **and** the Secretary of State is satisfied that that person will after the registration have or acquire some citizenship or nationality other than British citizenship, **then**, subject to subsection (4), the Secretary of State shall cause the declaration to be registered.

At the time of our formalisation of the British Nationality Act, our representation of the provisions of the Act was performed intuitively in an ad hoc manner. It is now possible to see that our treatment of exceptions was an instance of a general transformation. Other examples of negation in legislation and its elimination by means of transformation can be found in Ref. 8).

Before defining the transformation more precisely and proving that it preserves the semantics of the original program, we define our notation and semantics.

§3 The Language and Its Semantics

Our semantics for rules and exceptions is a modification of the answer set semantics of Gelfond and Lifschitz for extended logic programs. Therefore it is convenient to present the ordinary (unmodified) answer set semantics first.

3.1 Answer Set Semantics of Extended Logic Programs

An **extended logic program**[5] is a set of formulae of the form

$$L_0 \leftarrow L_1, ..., L_m, \text{ not } L_{m+1}, ..., \text{ not } L_n,$$

where $n \geq m \geq 0$, and each L_i is a literal. In this paper, formulae of the above form are called **clauses**. A **literal** is a formula of the form A or $\neg$A, where A is an atom.

A clause containing variables is treated as standing for the set of all ground instances of that clause. Therefore it is sufficient to define the notion of answer set for ground clauses. The definition is in two parts. First we consider the case where the program does not contain negation by failure. Then we consider the general case.

Let $\prod$ be any set of ground clauses not containing "not", and let Lit be the set of all ground literals in the language of $\prod$. The **answer set** of $\prod$, denoted by $\propto(\prod)$, is the smallest subset S of Lit such that

(1) for any clause $L_0 \leftarrow L_1, ..., L_m$ in $\prod$, if $L_1, ..., L_m \in S$, then $L_0 \in S$, and
(2) if S contains a pair of complementary literals, then S = Lit.

In case (2) we say that $\prod$ is **contradictory**.

Now let $\prod$ be any extended logic program not containing variables. Let Lit be defined as before. For any subset S of Lit, let $\prod^S$ be the set of clauses without "not" obtained

(1) by deleting every clause containing a condition "not L", with $L \in S$, and
(2) by deleting in the remaining clauses every condition "not L" if $L \notin S$.

S is an **answer set** of $\prod$ if and only if S is the answer set of $\prod^S$, i.e. if and only if $S = \propto(\prod^S)$. If $\prod$ is a general program (i.e. $\prod$ does not contain classical negation), then the definition of answer set coincides with the definition of stable model given by Gelfond and Lifschitz in Ref. 4).

A general logic program may have more than one stable model and therefore more than one answer set. For example, the program

$\prod_1$: $P \leftarrow \text{not } Q$
$Q \leftarrow \text{not } P$

has two answer sets

$\{P\}$ which is also the answer set of $\prod_1^{\{P\}} = \{P\}$, and
$\{Q\}$ which is also the answer set of $\prod_1^{\{Q\}} = \{Q\}$.

A general logic program may have no answer sets. For example,

$\prod_2$: $P \leftarrow \text{not } P.$

Similarly an extended logic program may have more than one answer set or no answer sets. Moreover, an extended logic program may also be **contradictory**. For example,

$\prod_3$: P
$\neg P.$

The answer set semantics treats clauses $L \leftarrow K$ as inference rules

from K
derive L

inhibiting use of the contrapositive, so that the inference

from $\neg L$
derive $\neg K$

is not allowed. Similarly, inferences such as

from $L \leftarrow K$
and $\neg L \leftarrow K$
derive $\neg K$

are also not allowed.

3.2 Answer Sets for Rules and Exceptions

To represent rules and exceptions we use the syntax of extended logic programs, but change the semantics. The syntactic difference between rules and exceptions is that rules have positive literals in the conclusion, whereas exceptions have classically negated literals in the conclusion. In the sequel we call theories containing rules and exceptions **logic programs with exceptions**, and sometimes just programs when the context makes the intended meaning clear.

The intuition behind reasoning with rules and exceptions is that exceptions have a higher priority than rules. That is if a contradiction arises between a rule and an exception, the exception overrides the rule. To capture this intuition, we modify the ordinary answer set semantics so that potential contradictions between rules and exceptions are avoided by preferring classically negated literals $\neg L$ to their positive counterparts L. This is done by changing the definition of $\prod^S$, by adding a third provision for deleting any rule in $\prod$ having conclusion L, with $\neg L \in S$.

More formally we define a new set ${}^S\prod$ in place of $\prod^S$. As before we assume that $\prod$ is variable-free. For any subset S of Lit, let ${}^S\prod$ be the set of clauses without "not" obtained

(1) by deleting every clause containing a condition "not L", with $L \in S$, and
(2) by deleting in the remaining clauses every condition "not L", if $L \notin S$, and
(3) by deleting every rule having a positive conclusion L, with $\neg L \in S$.

Clearly ${}^S\prod$ differs from $\prod^S$ only in the extra condition (3).

For variable-free programs $\prod$ not containing "not", let $\propto(\prod)$ denote the answer set of $\prod$, as defined for ordinary answer set semantics. Then, for any variable-free program $\prod$, S is an **e-answer set** if and only if $S = \propto({}^S\prod)$.

We show in Theorem 2, below, that e-answer sets never contain contradictions. Thus condition (2) of the definition of $\propto$ is unnecessary for e-answer sets. In Theorem 1, we show that if an extended logic program has a non-contradictory answer set, then that answer set is also an e-answer set. As a corollary, for logic programs without classical negation, stable models, answer

sets, and e-answer sets coincide.

Examples
The extended logic program

$\prod_3$: P
$\neg P$

which is contradictory according to the answer set semantics, has one e-answer set

$\{\neg P\}$.

The following example shows that rules and exceptions can be used for default reasoning.

$\prod_4$: Rules
pacifist ← quaker
hawk ← republican
quaker
republican
Exceptions
¬pacifist ← hawk
¬hawk ← pacifist

$\prod_4$ has two e-answer sets

S1 = {quaker, republican, pacifist, ¬hawk}
S2 = {quaker, republican, hawk, ¬pacifist}.

The two e-answer sets correspond to two different extensions in Reiter's default logic.[12] Extension S1 can be eliminated by removing the second exception. S2 can be eliminated by removing the first.

Theorem 1
If $\prod$ is a variable-free extended logic program and S is a non-contradictory answer set for $\prod$, then S is an e-answer set for $\prod$.

Proof
Assume S is a non-contradictory answer set for $\prod$. Then $S = \propto(\prod^S)$. But

$${}^S\prod = \prod^S - \{L \leftarrow L_1, ..., L_m \mid \neg L \in S\}.$$

Because S is non-contradictory, for any clause C such that

$$C = L \leftarrow L_1, ..., L_m \in \prod^S - {}^S\prod,$$

$L \notin S$.

But then C makes no contribution to $\propto(\prod^S)$. Therefore $\propto(\prod^S) = \propto({}^S\prod) = S$. So S is an e-answer set for $\prod$. □

Theorem 2
Let $\prod$ be a variable-free logic program with exceptions. Then $\prod$ is non-contradictory in the sense that it has no e-answer set S containing contradictory literals.

Proof
Suppose, on the contrary, that S is an e-answer set containing contradictory literals. Then S must contain a pair of literals L and $\neg$L such that ${}^{S}\prod$ contains two clauses

$$L \leftarrow K$$
$$\neg L \leftarrow K'$$

where all the literals in K and K′ belong to S.
However, by provision (3) in the definition of ${}^{S}\prod$, because $\neg L \in S$, ${}^{S}\prod$ cannot contain any clauses with conclusion L. So S cannot contain contradictory literals. □

§4 The Transformation

We define the transformation separately for the ground case, where the program contains no variables, and for the general case, where it does.

4.1 The Ground Transformation

The **transformation** is easy to define for programs $\prod$ containing no variables:

> for any rule in $\prod$ with conclusion L, if $\prod$ contains an exception with conclusion $\neg$L, then add not $\neg$L to the conditions of the rule.

For example, program $\prod_4$ is transformed into

pacifist ← quaker, not ¬pacifist
hawk ← republican, not ¬hawk
quaker
republican
¬pacifist ← hawk
¬hawk ← pacifist.

This can be simplified to

pacifist ← quaker, not hawk
hawk ← republican, not pacifist
quaker
republican
¬pacifist ← hawk
¬hawk ← pacifist.

This program has the same two e-answer sets as the original program $\prod_4$. Except for the last two clauses, this has the form of a conventional logic program, which can be executed by ordinary methods. Unfortunately, in this case SLDNF goes into an infinite loop when given the query ?pacifist or ?hawk. The abduction procedure of Eshghi and Kowalski,[3] on the other hand, computes both e-answer sets. However, it is not complete in the general case.

The transformation preserves the semantics of the original program, as shown by the following theorem.

Theorem 3
Let $\prod$ be a logic program with exceptions containing no variables, and let $\prod'$ be the extended logic program obtained from $\prod$ by applying the transformation. Then S is an e-answer set for $\prod$ if and only if S is an e-answer set for $\prod'$.

Proof
We shall show more generally that

$${}^{S}\prod = {}^{S}\prod', \text{ for any } S \subseteq \text{Lit}.$$

For any clause $C \in \prod$, let C′ denote the result of applying the transformation to C. Let ${}^{S}C$ and ${}^{S}C'$ denote the result of applying (1), (2), (3) of the definition of ${}^{S}\prod$ to the clauses C and C′ respectively. It suffices to show that ${}^{S}C$ and ${}^{S}C'$ are identical for all $C \in \prod$.
This is obviously the case when C = C′. It remains to consider the case where C has conclusion L and C′ differs from C in having an extra condition not $\neg$L. There are two subcases:

(a) $\neg L \in S$,
(b) $\neg L \notin S$.

In subcase (a), C is deleted because of provision (3) and C′ is deleted because of provision (1). So ${}^{S}C$ and ${}^{S}C'$ are identical.
In subcase (b), not $\neg$L is deleted from C′ by provision (2). After deletion of this condition from C′, the two clauses become identical. So ${}^{S}C$ and ${}^{S}C'$ are identical.
□

4.2 The General Transformation

For programs containing variables, we transform the rules by considering each exception in turn and transforming the rules with respect to each exception.

Let $\neg p(t') \leftarrow B$, be an exception and let $p(t) \leftarrow C$ be a rule, where t and t′ are vectors of terms and the two clauses have no variables in common. The set of all ground instances of the rule can be partitioned into two disjoint sets, one to which the exception applies, and one to which it does not. These two sets can be represented by general rules.

In the case where t and t′ do not unify, the exception is not applicable, the first set is empty, and the second set consists of all the ground instances of

the original rule. Therefore the **transformation** in this case leaves the original rule intact.

In the case where t is an instance of t′, the exception applies to all instances of the rule, and therefore the **transformation** replaces the original rule by the single rule

$$p(t) \leftarrow C, \text{ not } \neg p(t).$$

In all other cases, the rule is **transformed** into two rules. The first rule

$$[p(t) \leftarrow C, \text{ not } \neg p(t)]\theta$$

where θ is the most general unifier of t and t′, represents the set of all instances to which the exception applies. The second rule

$$p(t) \leftarrow C, \text{ not } \exists X \; t = t'$$

represents the set of all instances to which the exception does not apply. Here X is the vector of all variables occurring in t′. This second rule can be rewritten as two clauses by introducing a new predicate symbol, say unif:

$$p(t) \leftarrow C, \text{ not unif}(t)$$
$$\text{unif}(t').$$

These three cases complete the definition of the transformation for one rule with respect to one exception. To **transform** a logic program with exceptions in general, we successively transform each rule in the program with respect to each exception.

Note that, although we have presented the transformation as three separate cases in order to give it an intuitive justification, the first two cases can be regarded as special cases of the third.

Example

$\prod_5$: $p(X, a) \leftarrow q(X)$
$\neg p(b, Y) \leftarrow r(Y)$
$\neg p(c, Z) \leftarrow s(X, Z)$

The rule can be transformed with respect to the first exception, giving two rules:

$$p(b, a) \leftarrow q(b), \text{ not } \neg p(b, a)$$
$$p(X, a) \leftarrow q(X), \text{ not } X = b$$

which can be further transformed with respect to the second exception giving the following rules:

$$p(b, a) \leftarrow q(b), \text{ not } \neg p(b, a)$$
$$p(c, a) \leftarrow q(c), \text{ not } c = b, \text{ not } \neg p(c, a)$$
$$p(X, a) \leftarrow q(X), \text{ not } X = b, \text{ not } X = c.$$

As a further simplification, the second condition not c = b of the second rule can be deleted.

As mentioned earlier, two further transformations, renaming and macroprocessing, can be performed. Renaming eliminates classical negation and converts an extended logic program into a conventional logic program by introducing a new predicate symbol p′ for every negated predicate symbol p and replaces every occurrence of an atom $\neg p(t)$ by an atom p′(t). Gelfond and Lifschitz[5] have shown that such renaming respects the semantics of the original program. In many cases the new predicates can be eliminated from the conditions of clauses by macroprocessing as illustrated earlier.

The following theorem shows that the general transformation preserves the semantics of the original program.

Theorem 4

Let $\prod$ be a logic program with exceptions and let $\prod'$ be the result of applying the transformation to $\prod$. Then there is a one to one correspondence between e-answer sets of $\prod$ and e-answer sets of $\prod'$. Under this correspondence the two e-answer sets are identical, except for literals involving the new predicates introduced by the transformation.

Proof

It suffices to show that $\prod^{*\prime} = \prod'^{*\text{red}}$

where $\prod^*$ is the set of ground instances of $\prod$,
$\prod^{*\prime}$ is the result of applying the ground transformation to $\prod^*$,
$\prod'^*$ is the set of ground instances of $\prod'$, and
$\prod'^{*\text{red}}$ is the result of evaluating in $\prod'^*$ all conditions of the form "not unif(t)" where unif is a new predicate symbol introduced by the transformation, and then ignoring all clauses defining unif.

The proof of this equality is an elaboration of the justifications given to motivate the definition of the general transformation. The details of the proof are straightforward, and need not be presented here. □

§5 Comparison with Other Work

The e-answer set semantics presented in this paper is clearly an adaptation of the answer set semantics of Gelfond and Lifschitz. In their semantics, positive and classically negative literals have the same status, and contradictions can occur as a result. In our semantics, contradictions are avoided by giving preference to classically negative literals whenever they conflict with positive literals.

Thus, in the flying birds example, whereas the answer set semantics derives a contradiction from the assumption that John is both an ostrich and a bird, the e-answer set semantics avoids contradiction by deriving the negative conclusion that John does not fly in preference to the positive conclusion that he does.

Our treatment of rules and exceptions is in the spirit of Poole's[11] distinction between default rules and facts. In his semantics a default rule is assumed to hold provided it does not conflict with the facts. He does not discriminate between positive literals and classically negative literals, and he does not use negation by failure.

In our approach clauses with positive conclusions correspond to default rules. Exceptions, with negative conclusions, correspond to facts.

In contrast with Poole's treatment of default rules and facts as ordinary sentences of logic, the answer set and e-answer set semantics treat clauses as inference rules. Thus, for example, given the default rule

$$\text{fly}(X) \leftarrow \text{bird}(X)$$

and the fact

$$\neg\text{fly}(\text{john})$$

Poole would be able to derive the logical consequence

$$\neg\text{bird}(\text{john}),$$

whereas we would not. In order to inhibit this inference, Poole uses constraints, which are not necessary in our approach.

This difference between the semantics of clauses regarded as inference rules and the semantics of clauses regarded as sentences of logic does not arise in the ordinary logic programming case where there is no classical negation.

Treating clauses as inference rules makes our semantics for rules and exceptions similar to Reiter's default logic.[12] This is especially clear when the rules have been transformed with respect to the exceptions. A transformed clause of the form

$$L \leftarrow L_1, \ldots, L_m, \text{ not } \neg L$$

corresponds to the default

$$\frac{L_1, \ldots, L_m : ML}{L}$$

where ML means that L is consistent, or equivalently that $\neg L$ is not provable, i.e. not $\neg L$.

The semantics of a default theory are defined in terms of logical consequence in one of its extensions. Gelfond and Lifschitz[5] show that the answer sets for an extended logic program coincide with the extensions of the corresponding default theory.

An alternative approach, extending logic programming to deal with default reasoning, has been proposed by Pereira and Aparicio.[10] In our approach the semantics incorporates the assumption that exceptions have higher priority than the rules. In their approach priorities are expressed explicitly by

sentences in the theory.

This paper is a revision of an earlier report.[7] Because the report is discussed in the Gelfond and Lifschitz[5] paper, we outline here the main changes between this paper and the earlier report. Most of these changes are due to the simplifications made possible by using classical negation and by basing our semantics on answer sets.

In the earlier report we viewed exceptions as integrity constraints and presented the transformation as a way of eliminating integrity constraints from deductive databases. We were influenced in this point of view by the work of Asirelli et al.,[1] who proposed a transformation to eliminate certain restricted forms of integrity constraints from definite clause databases. Our transformation is an extension of theirs.

In the earlier report we outlined a proof theoretic proof of the equivalence between the original and the transformed programs. This proof required that the resulting program be locally stratified and that we identify a unique "retractable" condition in each integrity constraint in order to restore consistency. Thus, for example, we did not allow both

$$\neg P \leftarrow Q$$
$$\neg Q \leftarrow P$$

in the set of integrity constraints (exceptions). In this paper we have been able to give a simpler model theoretic proof of the equivalence, without the local stratification and retractibility restrictions.

Although, compared with the approach in our earlier report,[7] the modified answer set semantics significantly simplifies the proof of equivalence, it does so at the expense of complicating the semantics. It would be interesting to determine whether the proof can be simplified without changing the semantics.

The model theoretic techniques developed in this paper can be extended to deal with a number of more general cases. It is possible, for example, to associate exceptions with individual rules rather than with entire predicates. It is also possible to deal with rules having negative conclusions and exceptions having positive conclusions. The techniques can also be extended to cater for hierarchies of exceptions. Discussion of these extensions is beyond the scope of this paper.

Acknowledgements

This work was supported by the Science and Engineering Research Council and the Esprit Basic Research Project, Compulog. We are grateful to Hendrik Decker, Tony Kakas, John Lloyd, Luis Pereira, and the referees for helpful comments on earlier drafts of this paper.

References

1) Asirelli, P., De Santis, M. and Martelli, M., "Integrity Constraints in Logic Databases," *J. Logic Programming*, *2,3*. pp. 221-232, 1985.
2) Clark, K. L., "Negation as Failure," in *Logic and Databases* (Gallaire, H. and Minker, J., eds.), Plenum Press, pp. 293-322, 1978.
3) Eshghi, K. and Kowalski, R. A., "Abduction Compared with Negation by Failure," *Proc. of the Sixth International Logic Programming Conference*, MIT Press, 1989.
4) Gelfond, M. and Lifschitz, V., "The Stable Model Semantics for Logic Programs," *Proc. of the Fifth International Conference and Symposium on Logic Programming* (Kowalski, R. A. and Bowen, K. A., eds.), 2, pp. 1070-1080, 1988.
5) Gelfond, M. and Lifschitz, V., "Logic Programs with Classical Negation," *Proc. of the Seventh International Logic Programming Conference*, MIT Press, 1990.
6) Ioannides, A. J., "Transformation Algorithms for Representing Knowledge without Integrity Constraints," *M. Sc. thesis*, Department of Computing, Imperial College, London, 1989.
7) Kowalski, R. A. and Sadri, F., "Knowledge Representation without Integrity Constraints," Department of Computing, Imperial College, London, 1988.
8) Kowalski, R. A., "The Treatment of Negation in Logic Programs for Representing Legislation," *Proc. of the Second International Conference on Artificial Intelligence and Law*, pp. 11-15, 1989.
9) Lloyd J. W., *Foundatins of Logic Programming, 2nd extended edition*, Springer-Verlag, 1987.
10) Pereira, L. M. and Aparicio, J. N., "Default Reasoning as Abduction," *Technical Report*, AI Centre/Uninova, 2825 Monte da Caparica, Portugal, 1990.
11) Poole, D., "A Logical Framework for Default Reasoning," *Artificial Intelligence, 36*, pp. 27-47, 1988.
12) Reiter, R., "A Logic for Default Reasoning," *Artificial Intelligence*, *13*, pp. 81-132, 1980.
13) Sergot, M. J., Sadri, F., Kowalski, R. A., Kriwaczek, F., Hammond, P. and Cory, H. T., "The British Nationality Act as a Logic Program," *CACM, 29, 5*, pp. 370-386, 1986.