PROBLEMS AND PROMISES OF COMPUTATIONAL LOGIC

Robert A. Kowalski
Imperial College, Department of Computing,
180 Queen's Gate, London SW7 2BZ, UK

Abstract

The ultimate goal of the Basic Research Action, Compulog, is to develop the use of logic for all aspects of computation. This includes not only the development of a single logic for representing programs, program specifications, databases, and knowledge representations in artificial intelligence, but also the development of logic-based management tools. I shall argue that, for these purposes, two major extensions of logic programming are needed - abduction and metalevel reasoning. I shall also argue that extensions of full first-order logic may not be necessary.

0. Introduction

The term "computational logic" has no generally agreed meaning. Here I shall use the term, in the sense in which it is used in the ESPRIT Basic Research Action "Compulog", as the use of logic for all aspects of computing: not only for representing programs, program specifications, databases, and knowledge bases; but also for processing, developing, and maintaining them.

Historically, the techniques of computational logic have arisen from work on the automation of logical deduction, begun by logicians in the 1950s. This resulted during the 1970s in the development of efficient theorem-proving techniques, based on the resolution principle, for processing logic programs. More recent work has focussed on developing techniques for deductive databases and for default reasoning in artificial intelligence. Within Compulog we aim to develop unified logic-based techniques for knowledge representation, knowledge processing, and knowledge assimilation applicable to the three areas of programming, databases and artificial intelligence.

In this paper I shall present a personal view of some of the characteristic achievements and problems of computational logic. I shall touch upon some, but not all, of the work

areas of Compulog. I will focus in particular upon two major topics in Compulog: knowledge assimilation and metareasoning. I will not deal with three other Compulog topics: constraint logic programming, structured types, and program development, transformation and analysis. These other topics are addressed in other papers presented at the Symposium. I will also discuss the broader question of what subset of full first-order logic is needed for practical applications.

1. Knowledge processing, representation, and assimilation

Logic has traditionally focussed upon the problem of determining whether a given conclusion (or theorem) C is logically implied by a given set of assumptions (or theory) T. Computational logic in particular has been concerned with developing efficient theorem-proving methods for determining such implications. It has also been concerned with applying such theorem-provers as "knowledge processors", for theories representing logic programs, deductive databases, or knowledge bases, and for theorems representing procedure calls, database queries, or problems to be solved.

In computing we need to be concerned not only with knowledge processing, but also with knowledge representation and knowledge assimilation. Knowledge representation concerns the way in which knowledge is formalised. Because the same knowledge can generally be formalised in many different ways, the subject of knowledge representation is concerned with identifying useful guidelines for representation. In programming, for example, these guidelines deal with such matters as the choice of data structures, recursive versus iterative procedures, and the treatment or avoidance of side effects; in databases, with normal forms, nested relations and metadata; and in artificial intelligence, with such issues as the treatment of defaults and uncertainty.

Knowledge assimilation concerns the assimilation of new knowledge into an existing theory. The simplest case is that of an update to be added to a database. The update is rejected if it fails to satisfy certain integrity constraints. Otherwise it is inserted into the database. More elaborate cases of knowledge assimilation can occur if the theory has richer logical structure. Four cases stand out:

i) The update is the addition of a sentence which can be deduced from the existing theory. In this case the update can be ignored, and the new theory can remain the same as the old theory.

ii) The update is the addition of a sentence which together with some sentences in the existing theory can be used to derive some of the others. In this case the update can replace the derivable old sentences.

iii) The update can be shown to be incompatible with the existing theory and integrity constraints. In this case a belief revision process needs to determine how to restore satisfaction of the integrity constraints. This might non-deterministically involve rejection or modification of any of the assumptions which partake in the proof of incompatibility.

iv) The update is logically independent from the existing theory, in the sense that none of the above relationships (i), (ii) or (iii) can be shown. In this case the simplest way to construct the new theory is to insert the update into the old theory. In many situations, however, it may be more appropriate to generate an abductive explanation of the update and to insert the explanation in place of the update into the old theory. By the definition of abduction, the explanation is so constructed that the original update can be derived from the new theory and the new theory satisfies any integrity constraints. The derivation of such an abductive explanation might be non-deterministic, and result in several alternative new theories.

Thus knowledge assimilation is a significant extension of database updates. It can play a major role in the incremental development and maintenance of logic-based programs, program specifications, and knowledge bases.

The topics of knowledge processing, representation, and assimilation are interrelated, and will be addressed in the remaining three sections of this paper. Section (2), which deals with abduction and integrity constraints, is specifically concerned with knowledge assimilation, but is also concerned with procedures for performing abduction and integrity checking, as well as with representing defaults. Section (3) deals with metareasoning, which is an important technique for specifying and implementing both theorem-provers and knowledge assimilators. It also considers the use of metalogic for representing modalities such as knowledge and belief. Section (4) deals with the question of whether full first-order logic is necessary for knowledge representation, or whether extensions of logic programming form might be more useful. Although essentially a knowledge representation matter, this issue has important consequences for the kinds of theorem-provers and knowledge assimilators needed for practical applications.

2. Abduction and integrity constraints

Abduction seems to play an important role in everyday human problem solving. Charniak and McDermott, for example, in their "Introduction to Artificial Intelligence" [51] argue that abduction plays a key role in natural language understanding, image recognition and

fault diagnosis; Poole [42] argues that it can be used to give a classical account of default reasoning; Eshghi and Kowalski [14] argue that it generalises negation by failure in logic programming; and within Compulog, Bry [7] and Kakas and Mancarella [24] have shown that abduction can be used to solve the view update problem in deductive databases. Several authors [12, 14, 16] have shown that it can be implemented naturally and efficiently by a simple modification of backward reasoning.

Given a theory T, integrity constraints I, and a possible conclusion C, an abductive explanation of C is a set of sentences Δ such that

$T \cup \Delta$ logically implies C, and
$T \cup \Delta$ satisfies I.

The notion of integrity constraint arises in the field of databases as a property which a database is expected to satisfy as it changes over the course of time. As Reiter [44] explains in his contribution to this Symposium, different ways to formalise constraints and to define constraint satisfaction have been proposed. For present purposes, and for simplicity's sake, I shall assume that constraints are formulated as denials, and that constraint satisfaction is interpreted as consistency in a logic with an appropriate semantics for negation by failure. Such a semantics might be given, for example, either by the completion of the theory [9] or by stable models [17].

I shall also assume that the theory is represented as a set of clauses in logic programming form

$$A_0 \leftarrow A_1, \ldots, A_n$$

where the conclusion A_0 is an atomic formula and the conditions A_i, $1 \leq i \leq n$, are atomic formulae or negations of atomic formulae. Constraints are represented as denials of the form

$$\leftarrow A_1, \ldots, A_n$$

Problems to be solved also have the form of denials, but are more naturally expressed with the negation symbol written as a question mark.

$$? A_1, \ldots, A_n$$

2.1 Fault diagnosis

The following simplified example of some of the causes of bicycle faults shows that abduction is a natural procedure for performing fault diagnosis.

Theory:
wobbly-wheel ← flat-tyre
wobbly-wheel ← broken-spokes
flat tyre ← punctured-tube
flat tyre ← leaky-valve

Constraint:
← flat-tyre, tyre-holds-air

To determine the possible causes of wobbly-wheel, it suffices to seek the abductive explanations of wobbly-wheel. This can be performed simply by reasoning backward, logic programming style, from the goal

? wobbly-wheel.

Instead of failing on the subgoals

? punctured-tube
? leaky-valve
? broken-spokes

these subgoals can be assumed to hold as hypotheses, provided they are consistent with the theory and integrity constraint, which they are.

In this example the three subgoals correspond to three alternative hypotheses, which give rise in turn to three alternative new theories, each of which explains the conclusion. Given the lack of further information such multiple solutions are unavoidable.

Given the additional information

tyre-holds-air

however, the two hypotheses

punctured-tube
leaky-valve

become inconsistent and have to be withdrawn non-monotonically.

Fault diagnosis can also be performed by deduction alone, as it is in most current expert systems. However, this requires that the natural way of expressing laws of cause and effect be reversed. This can be done in two quite different ways, one using logic programming form, the other using non-Horn clauses.

In the bicycle faults example, ignoring the integrity constraint, the deductive, logic programming representation might take the form

possible(flat-tyre) ← possible(wobbly-wheel)
possible(broken-spokes) ← possible(wobbly-wheel)
possible(punctured-tube) ← possible(flat-tyre)
possible(leaky-valve) ← possible(flat-tyre)
possible(wobbly-wheel)
? possible(X)

Even ignoring the problem of representing the integrity constraint, compared with the abductive formulation, this formulation is lower-level - more like a program than like a program specification.

The alternative, non-Horn representation has the form

flat-tyre v broken-spokes ← wobbly-wheel
punctured-tube v leaky-valve ← flat-tyre
wobbly-wheel

In this formulation it is natural to reason forward rather than backward, deriving the disjunction

punctured-tube v leaky-valve v broken-spokes

of all possible causes of the fault. This formulation has some advantages, including the ease with which it is possible to represent integrity constraints. Given, for example, the additional sentences

← flat-tyre, tyre-holds-air
tyre-holds-air

it is possible monotonically to derive the new conclusion

broken-spokes.

The non-Horn formulation of the bicycle faults problem is an example of a general approach to abduction through deduction proposed by Console et al [10].

It also illustrates the phenomena, to be discussed in section (4) below, that many problems can be represented both in logic programming and non-Horn clause form. (In this case, it is easy to see that the non-Horn formulation is the only-if half of the completion of the Horn clause formulation.) My own intuition is that logic programming form is simpler and easier to implement, but that it entails the need for extensions such as abduction and integrity constraints. It may also be, however, that the non-Horn clause form is useful for certain purposes.

2.2 Default reasoning

Poole [42] was perhaps the first to argue that abduction provides a natural mechanism for performing default reasoning. This can be illustrated with Reiter's example [43] of conflicting defaults. Expressed as a logic program with abduction and integrity constraints this can be represented in the form

Theory: pacifist(X) ← quaker(X), normal-quaker(X)
hawk(X) ← republican(X), normal-republican(X)
quaker(nixon)
republican(nixon)

Constraint: ← pacifist(X), hawk(X)

It is possible to explain both the conclusion

pacifist(nixon) with the hypothesis
normal-quaker(nixon)

and the conclusion

hawk(nixon) with the hypothesis
normal-republican(nixon).

Each hypothesis on its own is consistent with the integrity constraint, but together the hypotheses are inconsistent. This phenomenon of conflicting defaults is sometimes thought to be a problem with default reasoning, but it is a natural characteristic of abduction. Moreover, as Poole [42] has shown, maximally consistent sets of abductive hypotheses correspond to extensions in Reiter's default logic.

The problem of accepting the existence of conflicting defaults seems to be partly psychological and partly technical. If default reasoning is viewed as deduction, then it is natural to view extensions in default logic as models, and to regard logical consequence as truth in all models. But proof procedures for default logic determine only what holds in a single extension, rather than what holds in all extensions. However, if default reasoning is viewed as abduction, then it is natural to view extensions as theories, and to view multiple extensions as alternative theories. In this case, proof procedures for determining what holds in single extensions are exactly what is needed.

The number of alternative abductive hypotheses can often be reduced by assigning priorities between them. Such priorities can be formulated as statements in the theory itself, as shown by Pereira and Aparicio [40].

2.3 Negation by failure

Negation by failure is a simple and efficient form of default reasoning in logic programming. It has been shown [14] that abduction simulates and generalises negation by failure. This can be illustrated most simply with a propositional example.

p ← not q
q ← r

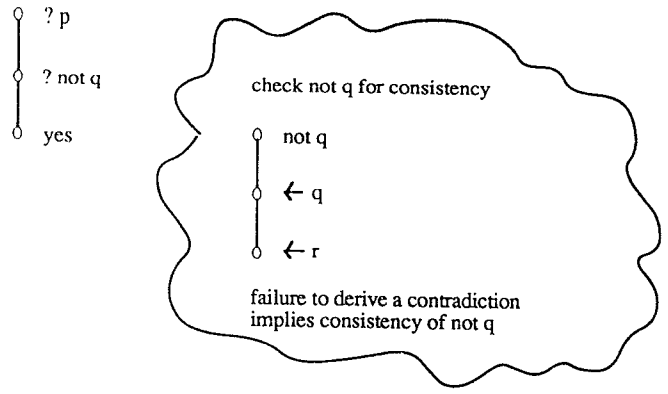
The negative condition can be thought of as a possible abductive hypothesis:

not q may be assumed to hold
if it is consistent to assume it holds.

That it is inconsistent to hold both an atom and its negation can be expressed by the integrity constraint

← q, not q

It is now possible to show p by means of backward reasoning with abduction and integrity checking, where not q is treated as a composite positive atom.



Here the possible hypothesis not q is tested by forward reasoning, as in the consistency method of integrity checking [45]. It is easy to see in this case that abduction with integrity checking simulates negation by failure.

The situation is more complex with nested negation in examples like

p ← not q
q ← not r

With only the two integrity constraints

← q, not q
← r, not r

The test for consistency of not q has two possible outcomes:

1) not q is inconsistent
with the hypothesis not r.
So p fails.

2) not q is consistent
without the hypothesis not r.
So p succeeds.

The first outcome accords with negation by failure, the second does not.

The second outcome can be eliminated by adding an extra integrity constraint

r v not r

which is viewed as a metalevel or modal epistemic statement [44] that

either r or not r
can be derived from the theory.

This eliminates the second outcome, because, since r cannot be proved, the hypothesis not r must be assumed. The atom r cannot be proved because, when abduction is used simply to simulate negation by failure, only negative atoms are assumed as hypotheses.

Under the interpretation of negative conditions as assumable hypotheses, abduction with appropriate integrity constraints simulates negation by failure. Moreover, there is a simple one-to-one correspondence between sets of abducible hypotheses satisfying the integrity constraints and stable models [14] of the original logic program.

Under the same interpretation, abduction with integrity checking generalises negation by failure. For a program such as

p ← not q
q ← not p

it gives the two alternative outcomes

1) p with the hypothesis not q
2) q with the hypothesis not p,

corresponding to the two stable models of the program. As in fault diagnosis and other applications of abduction, such multiple alternatives are quite acceptable.

2.4 The Yale Shooting Problem

This problem was identified by Hanks and McDermott [19] as an example of default reasoning where circumscription [36] and default logic [43] give intuitively unacceptable results. It is interesting to note that the example has the form of a logic program, and that negation by failure gives the correct result.

Here

t(P, S)

expresses that property P holds in situation S,

ab(P, E, S)

expresses that P is abnormal with respect to event E occurring in situation S, and the term

result(E, S)

names the situation that results from the occurrence of event E in situation S.

t(alive, s0)
t(loaded, result(load, S))
t(dead, result(shoot, S)) ← t(loaded, S)
t(P, result(E, S)) ← t(P, S), not ab(P, E, S)
ab(alive, shoot, S) ← t(loaded, S)

In circumscription, minimising the predicate "ab", the theory has two minimal models. In one model the atom

1) t(dead, result(shoot, result(wait, result(load, s0))))

is true. In the other the atom

2) t(alive, result(shoot, result(wait, result(load, s0))))

is true. Defining logical consequence as truth in all models means that using circumscription the disjunction of (1) and (2) is a consequence of the theory. In the corresponding formulation in default logic, (1) and (2) hold in alternative extensions.

Hanks and McDermott argue that the model in which (1) holds is not intuitively acceptable. In the body of literature which has since been devoted to the problem, few commentators have observed that negation by failure gives the intuitively correct result [2, 14, 15].

Under the interpretation of negation by failure as a special case of abduction with integrity constraints, it can be argued that what is missing in the circumscriptive and default logic

formulations of the problem is an appropriate analogue of the disjunctive integrity constraints [14]. Another abductive solution to the Yale shooting problem has been presented by [22].

2.5 Rules and exceptions

Closely related to abduction is another extension of logic programming in which clauses can have negative as well as positive conclusions. Clauses with positive conclusions represent general rules, and clauses with negative conclusions represent exceptions. This can be illustrated by the following formalisation of a familiar example.

Rules:
fly(X) ← bird(X)
bird(X) ← ostrich(X)
bird(X) ← penguin(X)
bird(tweety)
ostrich(ozzy)

Exceptions: ¬ fly(X) ← ostrich(X)
¬ fly(X) ← penguin(X)

The stable model semantics of Gelfond and Lifschitz [17] can be extended and the answer set semantics [18] modified to give a natural semantics for such extended logic programs [32]. In this example, under this semantics, it is possible to conclude

fly(tweety)
¬ fly(ozzy)

but not to conclude

fly(ozzy).

The semantics of rules and exceptions is related to the semantics of abduction in that, like abducibles, general rules hold by default, unless they are contradicted by exceptions, which behave like integrity constraints. This relationship is easy to see in the following reformulation of the quaker-republican example:

Rules:
pacifist(X) ← quaker(X)
hawk(X) ← republican(X)
quaker(nixon)
republican(nixon)

Exceptions: ¬ pacifist(X) ← hawk(X)
¬ hawk(X) ← pacifist(X)

Under the extended semantics it is possible to conclude

pacifist(nixon) in one "model", and
hawk(nixon) in another.

The different "models", however, are more like different abductive theories than they are like ordinary models. In particular, it is of greater relevance to discover what those "models" are and what holds in them, than it is to determine what holds in all of them.

Exceptions are more specialised than integrity constraints, in that they also indicate how to restore consistency in case they conflict with general rules. For example, compared with the constraint

← fly(X), ostrich(X)

the exception

¬ fly(X) ← ostrich(X)

indicates that in the case of a conflict between conclusions of the form

fly(t)
ostrich(t)

it is the former conclusion rather than the latter that should be withdrawn.

The two exceptions in the second formulation of the quaker-republican example correspond to the one integrity constraint in the earlier abductive formulation. It is also possible in the rules and exceptions formulation to have only one exception rather than two. The resulting representation would then have only one "model" rather than two.

It is possible to transform rules and exceptions into normal logic programs with negation by failure [32]. This is most easily performed in two stages. In the first stage extra conditions of the form

not ¬ A

are introduced into rules having conclusions of the form A, whenever there are exceptions with conclusions of the form ¬ A. (Appropriate account must be taken of the case where the atoms of the conclusions are unifiable rather than identical). In the second stage negated atoms of the form ¬ A are replaced by positive atoms with some new predicate symbol. This is easy to illustrate with the two examples above.

In the case of the first example, after the first stage we obtain the clauses

fly(X) ← bird(X), not ¬ fly(X)
bird(X) ← ostrich(X)
bird(X) ← penguin(X)
bird(tweety)
ostrich(ozzy)
¬ fly(X) ← ostrich(X)
¬ fly(X) ← penguin(X)

Using a new predicate symbol "abnormal" in place of "¬fly", after the second stage we obtain the normal logic program

fly(X) ← bird(X), not abnormal(X)
bird(X) ← ostrich(X)
bird(X) ← penguin(X)
bird(tweety)
ostrich(ozzy)
abnormal(X) ← ostrich(X)
abnormal(X) ← penguin(X)

The transformation preserves the meaning of the original formulation in the sense that the two formulations have the same "models".

In the case of the second example, after the two stages we obtain the program

pacifist(X) ← quaker(X), not ab1(X)
hawk(X) ← republican(X), not ab2(X)
quaker(nixon)
republican(nixon)
ab1(X) ← hawk(X)
ab2(X) ← pacifist(X)

This can be further simplified by "macro-processing" (or partially evaluating) the new predicates.

pacifist(X) ← quaker(X), not hawk(X)
hawk(X) ← republican(X), not pacifist(X)
quaker(nixon)
republican(nixon)

The resulting program (after one further simplification step) has the familiar form

p ← not q
q ← not p

and has the same two "models" as the original formulation.

2.6 Semantic Issues

The "model" theoretic semantics of rules and exceptions facilitates the proof that the transformation from rules and exceptions into normal logic programs preserves their meaning. It is also possible to generalise the stable model semantics of negation by failure to more general logic programs with abduction and integrity constraints [23]. It is an open problem whether such modifications of the semantics are really necessary.

As an alternative to modifying the semantics, it may be possible to view the meaning of these extensions of logic programming in metatheoretic terms. We have already remarked, in particular, that stable models behave in some respects more like theories than like conventional models. Moreover, our initial definition of abduction was formulated in metatheoretic rather than model theoretic terms.

Viewing the semantics of abduction and of rules and exceptions in terms of theories instead of models has the attraction that the underlying semantics remains that of the underlying deductive logic. This conservative approach to semantics has the advantage that seeming extensions of the logic are not extensions at all, but are defined in terms of metatheoretic relationships between object level theories. Just such a metatheoretic "semantics" for a special case of rules and exceptions can be found in the contribution by Brogi et al [6] presented at this Symposium. It has to be admitted, however, that extending this metatheoretic semantics to the general case does not seem to be entirely straightforward.

A related problem concerns the semantics of integrity constraints. In his contribution to the Symposium, Reiter [44] argues that integrity constraints should be understood as statements about what the theory "knows". I agree with this interpretation, but propose that knowledge be understood metatheoretically in terms of what can be proved from the theory. This is a topic for the next section.

3. Metareasoning

A metalanguage is a language in which the objects of discussion are themselves linguistic entities. A metalogic, moreover, is a metalanguage which is also a logic.

Many applications of logic are inherently and unavoidably metalogical - applications, which formalise proof procedures (also called metainterpreters) and knowledge assimilators, for example. Metalogic can also be used for applications which can be formalised using other logics, such as modal logics of knowledge and belief. In this section I shall discuss some of these applications and their implementations.

3.1 The demo predicate

For many of these applications it is useful to employ a two-argument proof predicate

demo(T, P)

which expresses that the theory named T can be used to demonstrate the conclusion named P. For other applications the first argument might be omitted or other arguments might be added - to name a proof of the conclusion or the number of steps in a proof, for example.

Perhaps the most common and most characteristic application of metalogic is to define and implement metainterpreters. Probably the simplest of these is a metainterpreter for propositional Horn clause programs.

(pr1) demo(T, P) ← demo(T, P ← Q),
demo(T, Q)

(pr2) demo(T, P ∧ Q) ← demo(T, P),
demo(T, Q)

Here "∧" names conjunction and (without danger of ambiguity) "←" names "←".

Theories can be named by lists (or sets) of names of their sentences, in which case we would need an additional metaclause such as

demo(T, P) ← member(P, T)

together with a definition of the member predicate.

Execution of the metainterpreter, for given names of a theory and proposed conclusion, simulates execution of the object level theory and conclusion. For example, execution of the object-level program and query

p ← q, r
q ← s
r
s
?p

can be simulated by executing the metaquery

? demo([p ← q ∧ r, q ← s, r, s], p)

using the meta interpreter. Here, for simplicity's sake, I have not distinguished between atomic formulae and their names.

For some applications, such as the implementation of program transformation systems, for example, naming theories by lists is both appropriate and convenient. For other applications, however, it is cumbersome and inconvenient. In many simple cases, moreover, the theory argument can be eliminated altogether as in the so-called "vanilla" metainterpreter [20]. In the case of the simple object-level program given above, for

example, execution of the object-level could be simulated by adding metalevel clauses to describe the object-level program.

demo(p ← q ∧ r)
demo(q ← s)
demo(r)
demo(s)

and posing the metalevel query

?demo(p)

using the same metainterpreter as before, but without the theory argument.

The vanilla metainterpreter uses different predicates for clauses which can be demonstrated because they are axioms (the predicate "clause") and clauses which can be demonstrated because they are derivable by means of one or more steps of inference (the predicate "solve"). I prefer the formulation presented above because of its greater generality and because the use of a single predicate emphasizes the similarity between the axioms of metalogic and the distribution axiom of modal logic.

Another curious feature of the vanilla metainterpreter is that it can be used for programs containing variables as well as for variable-free programs. This can be seen, for example, with the object-level program and query

p(X) ← q(X)
q(a)
q(b)
? p(Y)

which can be represented (incorrectly!) at the metalevel by

demo(p(X) ← q(X))
demo(q(a))
demo(q(b))
? demo(p(Y))

The representation is incorrect because the implicit quantifiers are treated incorrectly. The object-level clause is implicitly universally quantified

∀X[p(X) ← q(X)]

Instead of expressing that this universally quantified clause belongs to the program, the metalevel clause expresses

∀Xdemo(p(X) ← q(X))

namely that for every term t the clause

p(t) ← q(t)

belongs to the program. It expresses in particular that the clauses

p(a) ← q(a)
p(b) ← q(b)

belong to the program.

This explanation of the meaning of the metalevel clause also explains why the vanilla metainterpreter works for programs containing variables, even though it is, strictly speaking, incorrect. The representation of object level variables by metalevel variables implicitly builds into the representation of the axioms and of the conclusion to be proved the correct rules

(pr3) demo(Q) ← demo(forall(X, P)),
substitute(X, P, Y, Q)

(pr4) demo(exists(X, P)) ← demo(Q),
substitute(X, P, Y, Q)

where the predicate

substitute(X, P, Y, Q)

expresses that Q is the expression that results from substituting the term Y for the variable X in P. Given a correct metalevel representation, for example:

demo(forall(var(1), p(var(1)) ← q(var(1))))
demo(q(a))
demo(q(b))
? demo(exists(var(2), p(var(2))))

the (otherwise) incorrect metalevel representation can be derived by reasoning forward with the rule (pr3) and backward with the rule (pr4). This is because when the condition

substitute(X, P, Y, Q)

is resolved the variable Y is uninstantiated and consequently occurs uninstantiated also in the "output" Q.

Notice that in the correct metalevel representation, object-level variables are presented by variable-free terms - in this case by the terms var(1) and var(2). Naming object-level variables by variable-free terms at the metalevel is standard logical practice, and is featured in the amalgamated logic of Bowen and Kowalski [5] and in the metalogical programming language Gödel [8].

3.2 Meta for knowledge assimilation

Although there are situations where it is possible to omit the theory argument from the demo predicate altogether, there are other situations where the theory argument plays an essential role. This is illustrated in the following (simplified) representation of knowledge assimilation. Here the predicate

assimilate(T, Input, NewT)

expresses that a new theory named NewT results from assimilating an input sentence named Input into a current theory named T. The four clauses correspond approximately to the four cases of knowledge assimilation presented in section (1).

assimilate(T, Input, T) ← demo(T, Input)
assimilate(T, Input, NewT) ← T ≡ S∧T',
demo(T'∧Input, S),
assimilate(T', Input, NewT)
assimilate(T, Input, T) ← demo(T∧Input, □)
assimilate(T, Input, NewT) ← NewT ≡ T∧Δ,
demo(NewT, Input)

Here in contrast with [5] and [6] I have used the conjunction symbol to combine a theory (regarded as a conjunction of sentences) with a sentence. The infix function symbol ≡ can be regarded as expressing logical equivalence. This gives an elegant (but potentially inefficient) way of expressing in the second clause that

S∈T

where T is regarded as a set of clauses. The symbol □ in the third clause denotes logical contradiction. For simplicity's sake this clause caters only for the case where the input is rejected when it is inconsistent with the current theory. The fourth clause formalises a simplified form of abduction without integrity checking.

3.3 Meta for theory construction

Knowledge assimilation can be thought of as a special case of constructing new theories from old ones. Such more general theory construction is an important part of software engineering, promoting modularity, reuse of software, and programming-in-the-large. Theory construction, by its very nature, is essentially a metalinguistic operation. Applications of metalogic to theory construction are presented by Brogi et al [6] in their contribution to this Symposium.

As pointed out in [6] programming by rules and exceptions can be regarded as combining one theory "Rules" of rules together with another theory "Except" of exceptions. In the special case where the theory of exceptions consists only of unit clauses, or is logically equivalent to unit clauses, the combination has an especially simple definition. Here the function symbol "combine" names the resulting new theory

demo(combine(Rules, Except), ¬ P) ← demo(Except, ¬ P)
demo(combine(Rules, Except), P) ← demo(Rules, P),
not demo(Except, ¬ P)

In the general case, where the theory of exceptions is defined in terms of predicates defined by the theory of rules, the metatheoretic definition is more complex, and begins to approximate the model theoretic definition.

3.4 Reflection

In those cases where the theory and proposed conclusion are fully given, it is possible to execute calls to the demo predicate without using a metainterpreter, by using the object level theorem-prover directly instead. This corresponds to the use of a "reflection" rule of inference

$$\frac{T \vdash P}{demo(T', P')}$$

where T' and P' name T and P respectively. T' and P' must not contain (meta) variables, because otherwise it would not be possible to (correctly) associate unique T and P with T' and P'.

For example, given a call of the form assimilate(T, Input, NewT), where T and Input are fully instantiated, the reflection rule can be used to execute the induced calls to the demo predicate in the first three clauses of the simplified definition of assimilate. It cannot be used, however, to execute the call to demo in the fourth clause, because there the term NewT in the first argument of demo contains the uninstantiated variable Δ.

The metainterpreter, on the other hand, can be executed with any pattern of input-output. Thus a metalevel definition of an object-level theorem-prover is more powerful than the object-level theorem-prover itself. As we have just seen, for example, the object level theorem-prover might be able only to perform deduction, whereas the metainterpreter would be able to perform abduction as well.

Because the metalevel is generally more powerful than the object-level, the converse reflection rule

$$\frac{demo(T', P')}{T \vdash P}$$

is also useful, especially if the metainterpreter employs more sophisticated theorem-proving techniques than the object level proof procedure. Such reflection rules (also called "attachment" rules) were first introduced by Weyhrauch [50] in FOL. They are the sole means of metareasoning in the metalogic programming language proposed by Costantini and Lanzaroni [11].

It is important to realise that reflection rules are weaker than reflection axioms:

$$demo(T', P') \leftrightarrow T \vdash P$$

Tarski [48] showed that reflection axioms lead to inconsistency. Montague [38] and Thomason [49] showed that weaker forms of reflection axioms can also lead to inconsistency.

3.5 Theories named by constants

It is often inconvenient or inappropriate to name theories by lists of names of sentences, and more useful to name them by constant symbols. Thus, for example, we might use a constant symbol, say t1, to name the theory

$p \leftarrow q, r$
$p \leftarrow s$
r
s

by asserting the metalevel clauses

$demo(t1, p \leftarrow q \wedge r)$
$demo(t1, q \leftarrow s)$
$demo(t1, r)$
$demo(t1, s)$

The object-level query

$? p$

can then be represented simply by the metalevel query

$?demo(t1, p)$

An especially important theory is the definition of the demo predicate itself. This too can be named, like any other theory, by a list or by a constant symbol. Using a constant symbol, such as "pr", and restricting ourselves for simplicity to the propositional Horn clause case, we need to assert the metametalevel axioms

(pr5) $demo(pr, demo(T, P) \leftarrow demo(T, P \leftarrow Q) \wedge demo(T, Q))$
(pr6) $demo(pr, demo(T, P \wedge Q) \leftarrow demo(T, P) \wedge demo(T, Q))$

to express that pr contains the two axioms pr1 and pr2.

Notice that, as before, I have simplified notation and not distinguished, for example, except for context, between the predicate symbol demo and the function symbol demo, which is its name.

The clauses pr5 and pr6 do not assert explicitly that pr1 and pr2 are the only axioms of pr. Moreover, it is not clear whether or not the axioms pr5 and pr6 themselves should be axioms of pr. Assuming that they should, we would then need to assert sentences such as

(pr7) demo(pr, pr5)
(pr8) demo(pr, pr6)

where, for simplicity's sake, I have used the symbols pr5 and pr6 in place of the longer names. But then, for consistency's sake, we should also include pr7 and pr8 in pr, and so on, ad infinitum. Fortunately, if we choose to follow this route, there is a simple solution. Simply add to pr the single additional axiom

(pr9) demo(pr, demo(pr, P) ← P)

We may then take pr to consist of the axioms pr1, pr2, pr5, pr6, and pr9. We can then derive

demo(pr, demo(pr, P)) ← demo(pr, P)

by pr1 and pr9. Using this, we can derive pr7 from pr5, pr8 from pr6, and so on.

Notice that the notation of pr9 is not entirely precise. This could be remedied by employing an axiom schema

demo(pr, demo'(pr', p') ← p)

in place of pr9, where demo', pr', and p' name demo, pr, and p respectively, and p is any sentence name. Alternatively, we could use an axiom

demo(pr, demo'(pr', P') ← P) ← name(P, P')

where

name(P, P')

is a metalevel predicate relating names P to their names P'.

3.6 Meta for representing knowledge and belief

Interpreting "demo" as "believe", the axioms pr1, pr2, pr5, pr6, and pr9 of pr resemble modal formulations of belief. If naming conventions are carried out conventionally and precisely, then the metatheoretic formulation is syntactically more cumbersome than the modal formulations. Moreover, the metatheoretic formulation is potentially subject to the inconsistencies discovered by Montague [38] and Thomason [49]. Their results, however, do not apply directly to our formalisation.

Bearing in mind these cautions, the metatheoretic formulation, nevertheless, offers several advantages over the modal formulation. It avoids, in particular, one of the biggest weaknesses of the standard modal approach, namely that logically equivalent formulae can be substituted for one another in any context. This gives rise, among other problems, to the problem of omniscience: that an agent believes (or knows) all logical consequences of its explicitly held beliefs (or knowledge).

As Konolige [29] points out, interpreting belief as provability allows resource limitations to restrict the implicit beliefs that can be accessed from explicitly held beliefs. Resource limitations can be captured in metatheoretical formulations by adding an extra parameter to the demo predicate to indicate the number of steps in a proof. For example

(pr1') demo(T, P, N + M) ← demo(T, P ← Q, N),
demo(T, Q, M)

(pr2') demo(T, P ∧ Q, N + M) ← demo(T, P, N),
demo(T, Q, M)

The metatheoretic approach is also more expressive than the modal approach. Even such simple sentences as pr1 and pr2, which quantify (implicitly) over names T of theories and names P and Q of sentences, are not expressible in standard modal logics. In modal logic the analogues of pr1 and pr2 are axiom schemas. Des Revières and Levesque [13] show that it is exactly this greater expressive power of metalogic that accounts for the inconsistencies that can be obtained when modal logics are reformulated in metalogical terms. They show that the inconsistencies can be avoided if the metalogical formulations are restricted analogously to the modal formulations.

Most systems of metalogic, such as FOL [50] and Gödel [8], separate object language and metalanguage. For some applications, such as the formalisation of legislation or the representation of knowledge and belief, it is convenient to combine them within a single language. Such an amalgamation of object language and metalanguage for logic programming was presented by Bowen and Kowalski [5].

The two-argument demo predicate, with amalgamation and theories named by constants, seems to be especially convenient for representing multi-agent knowledge and belief. Suppose, for example that we use the constant symbol "john", depending on the context, to name both the individual John and the theory consisting of John's beliefs. That John believes that Mary loves him could then be expressed by the metasentence

demo(john, loves(mary, john))

That John *knows* Mary loves him could be expressed by the combination of object level and metalevel sentences

(j1) demo(john, loves(mary, john))
(j2) loves(mary, john)

Here, as earlier, where the context makes the intended distinctions clear, I use the same notation for sentences and their names. Notice that the two sentences j1 and j2 could belong to the theory named john, the theory named mary, or any other theory.

Multi-agent knowledge and belief can also be represented in a metalogic where object languages and metalanguages are separated, as in the formalisation of the wise man problem by Aiello et al [1]. In this formulation of the problem, reflection rules are used to link different object level theories with the metatheory. The formulation of the same problem by Kim and Kowalski [25] using amalgamation of object language and metalanguage is closer to modal solutions [27, 28].

Another application where amalgamation seems to be especially appropriate is the formalisation of legislation, where one act (represented by a theory) often refers to other acts. The latest British Nationality Act 1981 [21, 46], for example, refers frequently to consequences of the previous Nationality Acts and of the Immigration Act 1971. It also contains ingenious examples of theory construction, such as that on page 13 [21], where a person is defined to be a British citizen "by descent", if that person

> "... was a person who, under any provision of the British Nationality Acts 1948 to 1965, was deemed for the purposes of the proviso to section 5(1) of the 1948 Act to be a citizen of the United Kingdom and Colonies by descent only, or would have been so deemed if male".

3.7 Semantic Issues

Because of inconsistencies, such as Tarski's formalisation of the liar paradox, which can arise when object language and metalanguage are combined in the same language, most formal systems, such as FOL [50] and the proposed language Gödel [8] keep the languages separate. However, applications of amalgamation and of the analogous modal logics demonstrate the utility of combining the different levels of language in the same formalism. The recent work of des Revières and Levesque [13] and of Perlis [41] suggests that the semantic foundations of modal logic can also be used to underpin the amalgamation of object level and metalevel logic. Given the link that has been established between the semantics of negation by failure and the stable expansions of autoepistemic logic [17], it may be that some extension of the semantics of autoepistemic logic [39] might provide such an underpinning.

However, it may be, instead, that Gödel's method of formalising the proof predicate of arithmetic within arithmetic will provide a better semantics for amalgamation, as was the original intention in [5]. Moreover, as several authors [4, 29, 47] have shown, metalogic itself can provide a semantics for certain modal logics. In this approach the modal logics might be viewed as abstractions which avoid the naming complications of metalogic.

4. Is full first-order logic necessary?

It is well known that Horn clause logic is a basis for all computation. This means that any logic at all, with recursively enumerable axioms and effective rules of inference, can be implemented using a Horn clause metainterpreter, by means of rules such as

$$\text{demo}(T, P) \leftarrow \text{demo}(T, P \vee Q), \text{demo}(T, \neg Q)$$

for example. This means that knowledge that can be represented in any logic can be represented, indirectly at least, in Horn clause logic alone.

But such purely theoretical results alone are not satisfactory in practice, where we are interested in natural and direct representations of knowledge. It is for this reason in fact that in the earlier sections of this paper I have argued that Horn clause logic programming should be augmented with such extensions as abduction, integrity constraints, and metalevel reasoning. I now want to argue that the extension to full first-order logic may be unnecessary.

But first it is important to clarify that extended logic programming form, as identified by Lloyd and Topor [34] for example, includes the use of full first-order logic for queries, integrity constraints, and the conditions of clauses. This assymetric use of full first-order logic is exemplified most dramatically in the case of relational databases, where it has been found useful in practice to limit severely the form of assumptions in the database to variable-free atomic formulae, but to allow queries and integrity constraints in full first-order form.

Lloyd and Topor have shown that logic programs extended to allow full first-order queries, integrity constraints, and conditions of clauses can be transformed into normal logic programming form. The correctness of their transformation is based on interpreting the semantics of negation by failure as classical negation in the completion of the program. It is not entirely clear how correctness is affected if some other semantics is taken instead.

The starting point for my argument that unrestricted, full first-order logic may not be needed for knowledge representation is an empirical claim: Most published examples of the use of logic in computing and artificial intelligence use logic programming form or some modest extension of logic programming form, usually without explicitly recognising it. I have already referred to several such examples in section (2). The Yale shooting problem, for example, is formulated directly in logic programming form. The birds-fly example is expressed in one formulation directly as a logic program and in another formulation as a logic program with exceptions. Similarly, the quaker-republican example has natural formulations both as a logic program with abduction and integrity constraints and as a logic program with exceptions. Numerous other examples can be found in the literature.

Another major class of applications is the formalisation of legislation [33]. The form of natural language in which legislation is written has a marked similarity to logic programming form. It also exemplifies many of the extensions, such as exceptions having negative conclusions [31] and metalinguistic constructs, which are needed for other purposes.

Interestingly, the natural language formulation of legislation generally expresses only the if-halves of definitions, and only rarely uses the if-and-only-if form. I have not found a single example of a sentence with a disjunctive conclusion.

Examples of disjunctive conclusions can, however, be found elsewhere in computing and artificial intelligence. It seems that many of these examples can often usefully be reformulated in an extended logic programming form.

I have already mentioned in section (2) that the deductive, non-Horn formulation of abduction expresses the only-if half of the if-and-only-if completion of the explicitly abductive, Horn formulation. As Clark showed in the case of negation by failure [9], reasoning about all derivations determined by a Horn clause program often simulates reasoning with the only-if, non-Horn, half of the completion of the program.

Some disjunctions

$$P \vee Q$$

can often be reexpressed naturally as one or two clauses in logic programming form

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

In the event calculus [30], for example, we found it useful to express the disjunction

for all time intervals I1 and I2 and properties P,
if P holds maximally for I1
and P holds maximally for I2,
then I1 = I2 or I1 preceeds I2 or I2 preceeds I1

in the logic programming form

I1 = I2 ← P holds maximally for I1,
P holds maximally for I2,
not I1 preceeds I2,
not I2 preceeds I1.

We also found it useful to reexpress the existential quantifier in the sentence

for every event type E and property P,
there exists a time interval I, such that,
if E initiates P,
then E starts I
and P holds maximally for I

by means of a function symbol which names I as a function of E and P

E starts after(E, P) ← E initiates P
P holds maximally for after(E, P) ← E initiates P.

The extent to which it is natural to use logic programming for knowledge representation might also be related to arguments for using intuitionistic logic for mathematics. There are several parallels between extensions of logic programming and intuitionistic logic. To begin with, the Horn clause basis of logic programming is a sublanguage of both classical and intuitionistic logic, and negation by failure is a constructive form of negation, in the spirit of intuitionistic negation. Moreover, the lack of disjunctive conclusions and existential statements in logic programming is similar to the property of intuitionistic logic that

if a disjunction P v Q is provable
then P is provable or Q is provable, and

if an existential statement ∃XP(X) is provable then
P(t) is provable for some term t.

There are cases, however, where reasoning with disjunctive conclusions seems unavoidable, even when starting from a logic programming position. Such a case arises, for example, when trying to prove that a certain property of a deductive database holds, independently of specific data.

The following very simplified example comes from the Alvey Department of Health and Social Security demonstrator project [3]. The clauses of the theory represent possible legislation and associated socio-economic hypotheses.

receives(X, £20) ← adult(X), age(X) ≥ 65
receives(X, £25) ← adult(X), age(X) < 65, unemployed(X)
receives(X, wages(X)) ← age(X) < 65, adult(X), employed(X)
£25 ≤ wages(X) ← adult(X)
adequate-income(X) ← receives(X, Y), £20 ≤ Y
X ≤ Y ← X ≤ Z, Z ≤ Y

Given such a theory, it might be desirable to show that, as a consequence, every adult has an adequate income

adequate-income(X) ← adult(X)

Unfortunately, even though the theory and theorem can be expressed in logic programming form, the theorem cannot be proved without extra disjunctive assumptions, such as

age(X) < 65 v age(X) ≥ 65 ← adult(X)
employed(X) v unemployed(X) ← adult(X)

But contrary to first appearances, the problem does not necessary go beyond logic programming form. The additional assumptions can be viewed as integrity constraints that constrain the information that can be added to the theory, rather than as statements belonging to the theory. Nonetheless, the use of a non-Horn clause theorem-prover, such as Satchmo [37], to solve the problem seems to be unavoidable. How this use of an object-level formulation of integrity constraints can be reconciled with arguments for a metalevel or modal formulation has still to be resolved.

Conclusion

I have argued in this paper that certain extensions of logic programming are needed for knowledge representation and knowledge assimilation. These extensions include abduction, integrity constraints, and metalevel reasoning. On the other hand, I have also argued that other extensions may not be needed - extensions such as modal logic and full first-order logic.

I have touched upon some of the unresolved semantic issues concerning these extensions, arguing for a conservative approach, which builds upon and does not complicate the semantics of the underlying logic. But these are complicated technical matters, about which it is good to have an open mind.

References

[1] Aiello, L. C., Nardi, D. and Schaerf, M. [1988]: "Reasoning about Knowledge and Ignorance", Proceedings of the FGCS, pp. 618-627.

[2] Apt, K. R., and Bezem, M. [1990]: "Acyclic programs", Proc. of the Seventh International Conference on Logic Programming, MIT Press, pp. 579-597.

[3] Bench-Capon, T.J.M. [1987]: "Support for policy makers: formulating legislation with the aid of logical models", Proc. of the First International Conference on AI and Law, ACM Press, pp. 181-189.

[4] Boolos, G. [1979]: The Unprovability of Consistency, Cambridge University Press.

[5] Bowen, K. A. and Kowalski, R. A. [1982]: "Amalgamating Language and Metalanguage in Logic Programming", in Logic Programming (Clark, K.L. and Tärnlund, S.-Å., editors), Academic Press, pp. 153-173.

[6] Brogi, A., Mancarella, P., Pedreschi, D., Turini, F. [1990]: "Composition operators for logic theories", Proc. Symposium on Computational Logic, Springer-Verlag.

[7] Bry, F., "Intensional updates: abduction via deduction". Proceedings of the Seventh International Conference on Logic Programming, MIT Press, pp. 561-575.

[8] Burt, A. D., Hill, P. M. and Lloyd, J. W. [1990]: "Preliminary Report on the Logic Programming Language Gödel. University of Bristol, TR-90-02.

[9] Clark, K. L. [1978]: "negation by failure", in "Logic and databases", Gallaire, H. and Minker, J. [eds], Plenum Press, pp. 293-322.

[10] Console, L., Theseider Dupré, D., and Torasso, P. [1990]: "A completion semantics for object-level deduction", Proc. AAAI Symposium on Automated Abduction, Stanford, March 1990.

[11] Costantini, S. and Lanzarone, G. A. [1989]: "A metalogic programming language", Proc. Sixth International Conference on Logic Programming, pp.218-233.

[12] Cox, P. T. and Pietrzykowski, T. [1986]: "Causes for Events: Their Computation and Applications", in Proceedings CADE-86, pp 608-621.

[13] des Rivières, J. and Levesque, H. J. [1986]: "The Consistency of Syntactical Treatments of Knowledge", in Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge, (Halpern, J. editor), pp. 115-130.

[14] Eshghi, K. and Kowalski, R. A. [1989]: "Abduction compared with negation by failure", Proceedings of the Sixth International Logic Programming Conference, MIT Press, pp. 234-255.

[15] Evans, C. [1989]: "Negation-as-failure as an approach to the Hanks and McDermott problem", Proc. Second International Symposium on Artificial Intelligence, Monterrey, Mexico, 23-27 October 1989.

[16] Finger, J. J. and Genesereth, M.R. [1985]: "RESIDUE: A Deductive Approach to Design Synthesis", Stanford University Report No. CS-85-1035.

[17] Gelfond, M. and Lifschitz, V. [1988]: "The stable model semantics for logic programs", Proceedings of the Fifth International Conference and Symposium on Logic Programming, (Kowalski, R. A. and Bowen, K. A. editors), volume 2, pp. 1070-1080.

[18] Gelfond, M. and Lifschitz, V. [1990]: "Logic programs with classical negation", Proceedings of the Seventh International Conference on Logic Programming, MIT Press, pp. 579-597.

[19] Hanks, S. and McDermott, D. [1986]: "Default reasoning, non-monotonic logics, and the frame problem", Proc. AAAI, Morgan and Kaufman, pp. 328-333.

[20] Hill, P. M. and Lloyd, J. W. [1989]: "Analysis of metaprograms", In Metaprogramming in Logic Programming", (H.D. Abramson and M.H. Rogers, editors), MIT Press, pp. 23-52.

[21] H.M.S.O. [1981]: "British Nationality Act 1981", Her Majesty's Stationery Office, London.

[22] Kakas, A. C. and Mancarella, P. [1989], "Anomalous models and Abduction", in Proceedings of 2nd International Symposium on Artificial Intelligence, Monterrey, Mexico, 23-27 October 1989.

[23] Kakas, A. C. and Mancarella, P. [1990], "Generalised stable models: a semantics for abduction", Proceedings of ECAI 90, pp. 385-391.

[24] Kakas, A. C. and Mancarella, P. [1990], "Database updates through abduction". Proceedings of VLDB 90.

[25] Kim, J.-S., and Kowalski, R. A. [1990], "An application of amalgamated logic to multi-agent belief", Proceedings of Meta 90, Leuven University.

[26] Konolige, K. [1982]: "A first order formalization of knowledge and action for a multiagent planning system", in Machine Intelligence 10, (Hayes, J.E., Michie, D., Pao, Y. H., editors), Ellis Horwood.

[27] Konolige, K. [1982]: "Circumscriptive Ignorance", Proc. AAAI-82, pp. 202-204.

[28] Konolige, K. [1985]: "Belief and Incompleteness", in Formal Theories of the Commonsense World, (Hobbs, J. and Moore, R. C. editors), Ablex Pub. Corp., pp. 359-403.

[29] Konolige, K. [1986]: "A Deduction Model of Belief", Pitman Research Notes in Artificial Intelligence.

[30] Kowalski, R. A. and Sergot, M. J. [1986]: "A logic-based calculus of events", New Generation Computing, Vol. 4, No. 1, pp. 67-95.

[31] Kowalski, R. A. [1989]: "The treatment of negation in logic programs for representing legislation", Proceedings of the Second International Conference on Artificial Intelligence and Law, pp. 11-15.

[32] Kowalski, R. A. and Sadri, F. [1990], "Logic programs with exceptions", Proceedings of the Seventh International Conference on Logic Programming, MIT Press, pp. 598-613.

[33] Kowalski, R. A., Sergot, M. J. [1990]: "The use of logical models in legal problem solving", Ratio Juris, Vol. 3, No. 2, pp. 201-218.

[34] Lloyd, J. W. and Topor, R. W. [1984]: "Making Prolog more expressive", Journal of Logic Programming, Vol. 3, No. 1, pp. 225-240.

[35] Lloyd J. W. [1987]: "Foundations of logic programming", second extended edition, Springer-Verlag.

[36] McCarthy, J. [1980]: "Circumscription - a form of nonmonotonic reasoning", Artificial Intelligence, Vol. 26, No. 3, pp. 89-116.

[37] Manthey, R. and Bry, F. [1988]: "SATCHMO: A theorem prover implemented in Prolog", Proc. Nineth Int. Conf. on Automated Deduction (CADE).

[38] Montague, R. [1963]: "Syntactical Treatments of Modality, with Corollaries on Reflection Principles and Finite Axiomatizability", Acta Philosophic Fennica 16, pp. 153-167.

[39] Moore, R.C. [1985]: "Semantical Considerations on Nonmonotonic Logic in Artificial Intelligence, Artificial Intelligence 25, (1985), pp. 75-94.

[40] Pereira, L. M. and Aparicio, J. N. [1990]: "Default reasoning as abduction", Technical report, AI Centre/Uninova, 2825 Monte da Caparica, Portugal.

[41] Perlis, D. [1988]: "Language with Self-Reference II: Knowledge, Belief and Modality", Artificial Intelligence 34, pp. 179-212.

[42] Poole, D. [1988]: "A logical framework for default reasoning", Artificial Intelligence 36, pp. 27-47.

[43] Reiter, R. [1980]: "A logic for default reasoning", Artificial Intelligence, 13 (1,2), pp. 81-132.

[44] Reiter, R. [1990]: "On asking what a database knows", Proc. Symposium on Computational Logic, Springer-Verlag.

[45] Sadri, F. and Kowalski, R. A. [1987]: "A theorem proving approach to database integrity", In Foundations of deductive databases and logic programming (J. Minker, editor), Morgan Kaufmann, pp. 313-362.

[46] Sergot, M. J., Sadri, F., Kowalski, R. A., Kriwaczek, F., Hammond, P. and Cory, H. T. [1986]: " The British Nationality Act as a logic program", CACM, Vol. 29, No. 5, pp. 370-386.

[47] Smorynski, C. [1985], "Self-Reference and Modal Logic", Springer-Verlag, New York.

[48] Tarski, A. [1936]: "Der Wahrheitsbegriff in den formalisierten Sprachen", Studia Philosophia, Vol. 1, pp. 261-405. English translation "The concept of truth in formalised languages" in A. Tarski "Logic, Semantics, and Metamathematics, Clarendon, Oxford, 1956.