Reasoning with Conditionals in Artificial Intelligence

Robert Kowalski
Department of Computing
Imperial College London

18 October 2007

Reasoning with conditionals has been the focus of much research in Cognitive Psychology, but has attracted comparatively less attention in Artificial Intelligence. However, if you scratch below the surface, it soon becomes apparent that conditionals, in one form or another, are arguably the most common form of knowledge representation in Artificial Intelligence. They are employed in such diverse paradigms as production systems, logic programming, and Baysian networks; and much theoretical and practical experience has been accumulated with their use.

The purpose of this paper is to suggest that work on conditionals in Artificial Intelligence and in Cognitive Psychology is complementary and can be mutually beneficial to both areas. I will focus on the work in Artificial Intelligence and will argue in particular that abductive logic programming combines many of the diverse features of conditionals in other Artificial Intelligence paradigms.

Classical logic

It is tempting to identify conditionals with material implication in classical logic. However, in Artificial Intelligence, except for mathematical theorem-proving and computer program verification, classical logic[1] has had relatively few applications.

In the early 1970s, classical logic came under attack from proponents of the procedural representation of knowledge. Despite the development of logic programming (Kowalski, 1974) and the logic programming language Prolog (Colmerauer et al, 1973), which aimed to reconcile procedural and logical representations of knowledge, the general view in AI is still that purely declarative, logical formalisations are inadequate.

In addition to logic programming, other attempts to overcome the deficiencies of classical logic led to the development of non-monotonic logics. The simplest of these uses negation as failure in logic programming and has been used for many applications, including expert systems and the formalisation of common sense reasoning.

Production systems

Production systems address the problems of classical logic in a different way, mainly by restricting knowledge representation to a combination of *facts*, which are atomic sentences, and *condition-action rules*, which have the syntax of conditionals. They also restrict reasoning to a kind of modus ponens, called *forward chaining*. Production systems have been widely used in Artificial Intelligence, mainly for expert systems applications. They have also been used widely in cognitive psychology to model human thinking and problem-solving (Newell, 1973; Anderson and Bower, 1973).

[1] Classical logic has been used more successfully, however, in some other areas of computing: for example in database query languages and program specification languages.

A typical *production system* (Simon, 1999) involves a "declarative" memory consisting of atomic sentences, and a collection of procedures, which are *condition-action rules*:

> *If conditions C, then actions A.*

Condition-action rules are also called *production rules*, just plain *rules*, or *if-then rules.* The most obvious use of production rules is to implement stimulus-response associations. For example:

> *If it is raining and you have an umbrella, then cover yourself with the umbrella.*
> *If you have an essay to write, then study late in the library.*

The action part of a rule is often expressed as a command, or sometimes as a recommendation, to perform an action.

Production rules are executed by means of a cycle, which reads a current input fact, uses *forward chaining* (from conditions to actions) to match the fact with one of the conditions of a production rule, verifies the remaining conditions of the rule, and then derives the actions of the rule as candidates to be executed. If more than one rule is "triggered" in this way, then *conflict-resolution* is performed to decide which actions should be executed. Inputs and actions can be internal operations or can come from and be performed upon an external environment.

Conflict resolution can be performed in many ways, depending upon the particular production system language. At one extreme, actions can be chosen purely at random. Or they can be determined by the order in which the rules are written: the first rule to be triggered executed first. At the opposite extreme, actions can be chosen by means of a full scale decision-theoretic analysis, deriving the expected outcomes of actions, evaluating their utility and probability, and selecting one or more actions with highest expected utility.

Compared with classical logic, which has both a declarative, model-theoretic semantics and diverse proof procedures, production systems have no declarative semantics at all. Production rules look like logical implications, but they do not have the semantics of implications. Moreover, forward chaining looks like *modus ponens*, but it is not truth-preserving: Actions are only recommendations for action and are not necessarily true.

The similarity of production rules to implications is a major source of confusion. Some authors (Thagard, 1996) maintain that production systems enable *backward reasoning*, from actions that are goals to conditions that are sub-goals. MYCIN (Shortliffe, 1974), for example, one of the earliest expert systems, used backward reasoning for medical diagnosis. MYCIN was and still is described as a production system, but is better understood in more logical terms.

When production rules are used to implement stimulus-response associations, they normally achieve unstated goals implicitly. Such goals and the resulting goal-oriented behaviour are said to be *emergent*. For example, the emergent goal of covering yourself with an umbrella if it is raining is *to stay dry*. And the emergent goal of studying late in the library if you have an essay to write is *to complete the essay*.

Production systems that implement stimulus-response associations are an attractive model of evolutionary theory. Their ultimate goal, which is to enable an agent to survive and prosper in competition with other agents, is not explicit, but emergent.

Despite their natural use to implement stimulus-response associations, many applications of production systems use production rules explicitly to reduce goals to sub-goals:

> *If goal G and conditions C then make H a sub-goal.*

Indeed, in ACT-R (Anderson and Bower, 1973), this is the typical form of production rules used to simulate human problem solving in such tasks as the Tower of Hanoi.

Thagard, in his textbook *Mind, an Introduction to Cognitive Science* (Thagard, 1996), also draws attention to this goal-oriented use of production rules and claims that they cannot be represented in logical form. On page 45, he gives the following example of such a rule:

> *If you want to go home for the weekend, and you have the bus fare,*
> *then you can catch a bus.*

Here the "action" *you can catch a bus* is a recommendation. The action can also be expressed as a command:

> *If you want to go home for the weekend, and you have the bus fare,*
> *then catch a bus.*

The use of the imperative voice to express actions motives the terminology "conflict resolution" to describe the process of reconciling conflicting commands.

Logic programming

Whereas conditionals of a sort, in the form of production rules, are the main kind of knowledge representation in production systems, conditionals of a logical kind are the only kind of representation in logic programming. Logic programming has been used for many applications in both Artificial Intelligence and other areas of computing. However, except for the work of Stenning and van Lambalgen (2004, 2005, 2008), it has not been used as a descriptive model[2] human reasoning at all.

Normal logic programs are sets of conditionals:

> *If* B_1 *and … and* B_n *then H*

where the *conclusion H* is an atomic formula and the *conditions* B_i are *literals*, which are either atomic formulas or the negations of atomic formulas. All variables are implicitly universally quantified in front of the conditional. *Facts* are the special case where $n = 0$ and there are no variables. Conditionals in logic programs are also called *clauses*. *Horn clauses* [3] are the special case where all of the conditions are atomic formulae.

For example, the three clauses:

> *If you have the bus fare and you catch a bus and not something goes wrong,*
> *then you will go home for the weekend.*

---

[2] For logic programming as a normative model of problem solving, see (Kowalski, 1979).

[3] Horn clauses are named after the logician Alfred Horn who studied their model-theoretic properties,

If you have the bus fare and you catch a bus,
then you will go home for the weekend.
You have the bus fare.

are a clause, a Horn clause and a fact respectively.

Logic programs are executed by means of *backwards reasoning* (from conclusions to conditions), treating conditionals as goal-reduction procedures:

to show/solve H, show/solve B1 and … and Bn.

For example, backward reasoning turns the conditionals:

If you study late in the library then you will complete the essay.
If you have the bus fare and you catch a bus,
then you will go home for the weekend.

into the procedures:

To complete the essay, study late in the library.
To go home for the weekend, check that you have the bus fare, and catch a bus.

Because conditionals in normal logic programming are used only backwards, they are normally written backwards:

H if B_1 *and … and* B_n.

so that backward reasoning is equivalent to "forward execution" in the direction in which the conditional is written. The Prolog syntax for clauses:

H :- B_1 *,… ,* B_n.

is deliberately ambiguous, so that the clause can be read either as a conditional written backwards or as a goal-reduction procedure executed forwards.

In practice, expert logic programmers use both the declarative reading of clauses as conditionals, so that programs correctly achieve their goals, and the procedural reading, so that programs behave efficiently. However, it seems that few programmers achieve this level of expertise. Many programmers focus entirely on the declarative reading alone and are disappointed when their programs fail to run effectively. Other programmers focus entirely on efficiency and loose the potential benefits of the declarative reading.

This difficulty that many programmers have in combining the declarative and procedural reading of clauses may be due, in part, to the greater complexity of computation in logic programming, compared with more conventional programming languages.

Computation in logic programming = backward reasoning + negation as failure + search
(This section can be skipped on a first reading.)

Computation in logic programming generates a *search tree* whose *nodes* are labelled by goals and whose *branches* are possible execution paths. Every *goal* in the tree is a conjunction[4] of literals:

G_1 *and …* *and* G_m.

The *root* of the tree is labelled by the initial goal. A path is a *successful computation* if it ends in a node labelled by the empty set of sub-goals, where m=0. A path is a *failed computation* if it ends in a node with non-empty sub-goals to which no inference can be applied.

Given a node with a non-empty set of sub-goals, computation proceeds by first selecting a sub-goal G_i for solution. If there is more than one sub-goal that can be selected, then as with conflict-resolution in production systems a decision needs to be made, selecting only one. Any selection strategy can be used. Prolog selects sub-goals in the order in which they are written. However, more intelligent problem-solving strategies, such as choosing the most constrained sub-goal first, are also possible (Kowalski, 1979).

In the simple, propositional case[5], if G_i is an atomic formula that is identical to the conclusion H of a clause H *if* B_1 *and …* *and* B_n, then *backward reasoning* is applied, replacing the selected sub-goal by the conditions of the clause, obtaining a *child* of the node labelled by the new goal:

G_1 *and …* *and* G_{i-1} *and* B_1 *and …* *and* B_n *and* G_{i+1} *and …* *and* G_m.

If G_i is the negation *not* G of an atomic formula G, then *negation as failure* is applied, performing a subsidiary computation with the same program, but with the initial goal G. The selected sub-goal *not* G succeeds if G fails, and the *only child* of the node is labelled by the new goal:

G_1 *and …* *and* G_{i-1} *and* G_{i+1} *and …* *and* G_m.

Given a program and an initial goal, the search tree is generated, and its branches are explored to find a successful computation. Prolog uses depth-first search, using the order in which clauses are written, to determine the order in which branches are explored; but other search strategies, such as breadth-first, best-first and parallel search are also possible.

Negation as failure

Negation as failure makes logic programming a non-monotonic logic. For example, given the clauses:

An object is red if the object looks red and not the object is illuminated by a red light.
The object looks red.

then the consequence:

The object is red.

---

[4] In fact, with the aid of auxiliary definitions, any formula in first-order logic can be a goal. For example, the conditional goal *if A then B* can be rewritten as the goal *not C* with the auxiliary definition *C if B and not A* (where *C* is an auxiliary predicate not occurring in any other clauses).

[5] In the non-propositional case, the selected sub-goal G_i is unified with H, by finding a most general substitution θ of terms for variables that makes the two atoms identical, i.e. $G_i\,\theta = H\,\theta$. The unifying substitution θ is then applied to the new goal.

follows as a goal, because there is no clause whose conclusion matches the subsidiary sub-goal, *the object is illuminated by a red light*. However, given the new clause, *the object is illuminated by a red light*, the subsidiary sub-goal now succeeds and the top-level goal now fails, non-monotonically withdrawing the consequence *The object is red.*

Negation as failure can be understood and justified in two main ways (Kowalski, 1979, Chapter 11), both of which interpret all the clauses of a program in which a given predicate occurs in the conclusion as the *only* clauses that can be used in establishing that conclusion. This is also known as the *closed world assumption* (Reiter, 1978).

The simplest formalisation (Clark, 1978) of the closed world assumption treats these clauses in the object-language as the *definition* of the predicate in if-and-only-if form. If a predicate does not occur as the conclusion of any clause, then it is deemed to be *false.* This definition is called the *predicate completion* [6], and it gives negation as failure a truth-functional semantics. Consider, for example, the logic program:

you get help if you press the alarm signal button.
you get help if you shout loudly.

The completion of the program is:

you get help if and only if you press the alarm signal button or you shout loudly.
you press the alarm signal button if and only if false. (i.e. *not you press the alarm signal button.*)
you shout loudly if and only if false. (i.e. *not you shout loudly.*)

Given only these sentences, the completion logically implies the conclusion *not you get help*. However, the conclusion is not implied if the clauses are interpreted as uncompleted conditionals in classical logic.

The other main way of formalising the close world assumption is to interpret the phrase "*not G"* in the meta-language, literally as "*G cannot be shown", "G does not hold",* or *"G is not believed".* Applying this interpretation informally to the two clauses in the example above, the conclusion *you get help* can not be shown, because both of the conditions in the two alternative ways of showing the conclusion can not be shown using only the given program, and therefore the conclusion *not you get help* succeeds[7].

In most cases these two interpretations of the closed world assumption and negation as failure justify the same logical consequences. In many cases, the consequences differ from those

---

[6] Notice that the use of *if and only if* here does not make the two sides of the "equivalence" symmetric. The left-hand side is an atomic formula defined by the formula on the right-hand side.

[7] This meta-level interpretation is formalised by *expanding* the given program P with a set Δ of cannot-be-shown sentences that is *stable* in the sense that the set Δ is truly the set of sentences that cannot be shown from the expanded set $P \cup \Delta$, i.e. $\Delta = \{not\ G \mid G\ can\ not\ be\ shown\ from\ P \cup \Delta\}$. This formalisation gives negation as failure an *auto-epistemic* semantics (Gelfond and Lifschitz, 1990).

For example the stable expansion of the logic program P above is Δ = *{ not you press the alarm signal button, not you shout loudly, not you get help}*. The expanded set of clauses $P \cup \Delta$ trivially implies the conclusion *not you get help*.

sanctioned by classical logic, but they seem to be similar to those observed in psychological studies of human reasoning.

Abductive reasoning with logic programs

Backward reasoning, in the form of SLD-resolution (Kowalski, 1974), is a sound and complete inference rule for Horn clauses[8]. However, the closed world assumption justifies other inferences, such as negation as failure, as we have just seen.

Negation as failure is an integral feature of normal logic programming. However, the closed world assumption also sanctions other inferences, the most important of which is abduction. Abductive reasoning is a defining feature of abductive logic programming, as we will see later.

For example, given *only* the clause:

A if B

and then told that *A* holds, it follows that *B* must hold.

Abductive reasoning can be justified deductively in both interpretations of the closed world assumption: In the case of the clause above, it can be justified at the object level, because given *A if and only if B* and given also *A*, it follows that *B*. It can also be justified at the meta-level, because given *A if B* as the *only* way of showing *A* and given that *A* can be shown, it follows that it must also be possible to show *B*.

Notice that the closed world assumption also justifies *switching* the direction of the implication: From *A if B*, conclude *B if A*.

Consider now a similar situation where, however, we are given an additional way of concluding *A*. Given *only* the clauses:

A if B.
A if C.

and then told that *A* holds, it no longer follows that *B* holds, but that *B or C* holds instead.

It is also possible to *switch* the direction of the implication, concluding *B or C if A*. The disjunction in the conclusion of *B or C if A* can be eliminated by using negation as failure:

B if A and not C.
C if A and not B.

Switching the direction of conditionals is a common technique used certain applications, such, as fault diagnosis, where the direct representation uses conditions for causes and conclusions for effects. However, this necessitates the use of abduction to explain observations of faults using the causal model. Since abduction is a more powerful form of inference than deduction, it is common to use deduction with the switched form of the conditionals instead.

---

[8] Given any Horn clause program *P* and goal *G*, $P \cup G$ is unsatisfiable if and only if there exists an SLD derivation of the empty clause, obtained by reasoning backwards from the goal *G*.

The relationship with conditionals in natural language

As Politzer and Bonnefon (this volume) observe, many conditionals in natural language have a truncated form (A -> H) that, to be understood fully, needs to be put into a wider context of the form:

(A & A1 &...An1) or .....(B & B1 &...Bn2) or....(C & C1 & ...Cn3) -> H

In this wider context, there may be both additional, unstated conditions (A1 &...An1) and additional, unstated, alternative ways (B & B1 &...Bn2) or....(C & C1 & ...Cn3) of establishing the conclusion H.

But, as we have just seen, in logic programming this wider context is exactly one half of a definition, which, at the object level, can be expressed in if-and-only-if form:

(A & A1 &...An1) or .....(B & B1 &...Bn2) or...(C & C1 & ...Cn3) <-> H

In logic programming, such definitions (and the conditionals that are embedded in them) are used for many purposes, the most important of which is to express means-ends procedures. Such procedures can be used both to generate plans to achieve H as an effect of various alternative causes and conditions, including A. But they can also be used for predicting the likely effects of actions. Definitions can also be used to represent taxonomies, in which an abstract class, such as the class of animals, is defined in terms of less abstract classes, such as the classes of vertebrates and invertebrates.

It is not easy, and therefore not common, in natural language to express definitions in their complete if-and-only-if form; and it is more common to express them incrementally in conditional form. But this leads to a fundamental ambiguity, whether to understand a conditional as a truncated form of the if-direction of a definition, or to understand it as a truncated form of the only-if-direction.

First, consider the case of a conditional of the form (A -> H), which is a truncated form of one of the alternative clauses of the definition of H:

A & A1 &...An1 -> H.

Typically, the condition A is the most significant of the conditions (A & A1 &...An1) in the clause, in which case the conditional (A -> H) can be regarded as a *first approximation* of the complete clause. The additional conditions are either deemed to hold by *default* or are spelled out in successively more precise formulations of the clause. This characteristic of conditionals can be viewed as a generalisation of the *qualification problem*: How to suitably qualify a conclusion so that all its conditions are adequately specified. The qualification problem is one aspect of the notorious *frame problem* in Artificial Intelligence.

The frame problem is normally solved in logic programming and other approaches to non-monotonic reasoning by interpreting the conditional (A -> H) as a *default conditional*, which is deemed to hold provided there is no evidence to the contrary, and by rewriting it in the *strict* form:

H if A and not abnormal1

where *abnormal1* is defined by successive clauses:

abnormal1 if not A1 ….
abnormal1 if not An1.

It can also be rewritten in the abductive logic programming form:

H if A and normal1.

where the positive condition *normal1* is an abducible, "open" predicate, to which the closed world assumption does not apply and which is restricted by suitable integrity constraints. We will investigate the role of integrity constraints later in this paper.

In the meanwhile, it is useful to note that the abductive formulation provides a link with Bayesian networks (Pearl, 1988), in which assigning a probability *p* to a conditional (A -> H) is equivalent to assigning the same probability *p* to the abducible *normal1* in the clause *H if A and normal1.* David Poole (1993, 1997) has shown that abductive logic programs with such probabilistic abducible predicates have the expressive power of discreet Bayesian networks.

Now consider the case of a *switched* conditional of the form (H -> A), which is a truncated form of the only-if direction of the definition:

H -> (A & A1 &...An1) or .....(B & B1 &...Bn2) or...(C & C1 & ...Cn3)

The disjunction in the conclusion of the switched conditional can be eliminated by using negation as failure and introducing a new predicate, say *alternative*, to represent the alternative ways of establishing the conclusion H.[9] The resulting sentences have the logic programming form:

A if H and not alternative

alternative if B & B1 &...Bn2……
alternative if C & C1 & ...Cn3.

If the strict conditional *A if H and not alternative* is rewritten as a default conditional, it then takes the literally switched form:

(H -> A).

These observations about the relationship between natural language conditionals and if-and-only-if definitions may help to explain some of the problems that people have with conditionals in natural language. In particular, they show, not only that a declarative conditional (X -> Y) may be missing additional conditions, but also that it may correspond to different directions of a definition.

[9] Of course, there are similar clauses for the other conditions A1 &...An1 in the same alternative as A, as well as for the other conditions, B, C, etc.

Consider, for example, the ambiguous conditional:

If an object looks red, then it is red.

The further statement:

This object looks red.

"invites" the inference:

This object is red.

However, the direction of the causal relationship between looking red and being red suggests that the conditional is a switched form of a definition specifying the alternative conditions under which an object looks red:

(An object is red & possibly other conditions) or (possibly other alternatives) <->
the object looks red.

Under the closed world assumption, which is equivalent to the assumption that there are no other alternatives, the definition can be rewritten in the completed logic programming form:

An object looks red if-and-only-if it is red and not abnormal.

However, the additional conditional:

If an object is illuminated by red light, then it looks red.

contradicts this closed world assumption and forces it to be reformulated in the new form:

An object looks red if-and-only-if it is red and not abnormal
or it is illuminated by a red light and not abnormal'.

The invited inference no longer holds[10].

These observations about the ambiguity of conditionals have implications for the psychology of logic programming. Not only do programmers have difficulty with combining the procedural and declarative interpretations of clauses, but they almost certainly have at least equal difficulty with the declarative style of programming alone. This may be one of the reasons that many programmers abandon declarative programming altogether, in favour of a purely procedural programming style.

The relationship between logic programs and production rules.

By comparison with logic programming, production systems not only lack a declarative reading, but they have a simpler behaviour, which is both easier to understand and easier to control.

[10] Politzer and Bonnefon (2006) explain the reasoning in this example in argumentation theoretic terms. In the light of the argumentation theory of Dung (Bondarenko et al, 1997; Dung et al, 2006) their explanation can be viewed as a meta-theoretic variant of the logic programming explanation given here.

However, in many cases their behaviour is similar. In particular, Stenning and van Lambalgen (2004, 2005, 2008) have observed that forward chaining with production rules of the form:

If goal G and conditions C then add H as a sub-goal.

is identical in behaviour to backward reasoning with clauses of the form:

G if C and H.

The production rule and the clause both give rise to the same goal-reduction procedure:

To achieve G, show C, and achieve H as sub-goal.

The production rule represents the procedure *subjectively* (or *intentionally*), in terms of the causal effect of the goal *G* on an agent's state of mind. The clause views it *objectively* (or *extensionally*), in terms of causal effects in the agent's environment. However, in both cases, the direction of the conditional is from conditions that are causes to conclusions that are results of those causes. This switching of the direction of the conditional is related to the switching associated with *evidential conditionals*, *epistemic* and *inferencial conditionals* as noted by Pearl (1988), Dancygier (1998) and Politzer and Bonnefon (2006). It also seems to be related to the switching of direction associated with the transformation of abduction into deduction.

Viewed in these terms, Thagard's example of the production rule:

If you want to go home for the weekend, and you have the bus fare,
then you can catch a bus.

is a subjective representation of the goal-reduction procedure:

To go home, check that you have the bus fare, and catch a bus.

which can also be represented objectively by the logic programming clause:

you will go home for the weekend if
you have the bus fare and you catch a bus.

Notice that the causal relationship in both formulations of the procedure has as an implicit associated degree of uncertainty. In the case of the production rule, the uncertainty is associated with whether or not the agent's conflict resolution strategy will chose the action of the rule if other rules with other incompatible actions are triggered at the same time.

In the case of the clause, the uncertainty is associated with whether or not other un-stated conditions, which can be summarised as *nothing goes wrong*, also hold. This condition can be stated explicitly using negation as failure[11]:

you will go home for the weekend if
you have the bus fare and you catch a bus and

[11] The formulation without this extra condition can be regarded as an approximation to the fully specified clause. It is an approximation also because it is not sufficient to take any old bus to get home, but to take a bus that is on a route that goes home.

not something goes wrong.

This representation, using negation as failure, does not attempt to quantify or even to qualify the associated degree of uncertainty. However, in abductive logic programming it can be quantified by using instead a positive abducible predicate, say *the journey is successful,* and by associating a probability with the abducible predicate. For example:

you will go home for the weekend if
you have the bus fare and you catch a bus and
the journey is successful.

the journey is successful (with probability .995).
something goes wrong with the journey (with probability .005).

The relationship between clauses and production rules that are used to represent goal-reduction procedures leaves open the question of the relationship when production rules are used instead to represent stimulus-response associations.

For example, it is unnatural to interpret the condition-action rule:

If it is raining and you have an umbrella, then cover yourself with the umbrella.

as a procedure for solving the goal of covering oneself with an umbrella. It is similarly unnatural, therefore, to reformulate it as the corresponding clause:

you cover yourself with an umbrella if it is raining and you have an umbrella.

I will argue that such production rules are better understood as *conditional goals*, which can be interpreted as integrity constraints in abductive logic programming. Conditional goals are closer to conditionals in classical logic than they are to clauses in logic programs.

Conditionals in active deductive databases

Integrity constraints in abductive logic programming can perhaps most easily be understood in comparison with integrity constraints in databases, where they are used to prevent unacceptable updates. In active databases, integrity constraints can also be used to compensate for otherwise unacceptable updates, by allowing the updates, but performing corrective actions, to ensure that integrity is maintained.

Thus the condition-action rule for covering oneself when it rains can be understood as an active integrity constraint:

If it is raining and you have an umbrella, then you cover yourself with the umbrella.

which is identical in syntax to the condition-action rule, except for the fact that the action part is not expressed as a command or recommendation, but as a statement in declarative form. Unlike the original condition-action rule, which does not have a logical semantics, the integrity constraint has the semantics of a logical implication.

An agent could use such a constraint as a *maintenance goal* to maintain the integrity of its "knowledge base", in much the same way that an active database system uses integrity constraints

to maintain the integrity of its data. Integrity checking is initiated by forward reasoning, or modus ponens, when an observation that updates the agent's knowledge base matches a condition of a conditional integrity constraint. It proceeds by checking whether the other conditions of the constraint also hold, and if they do, it derives the conclusion of the constraint as an *achievement goal*.

An achievement goal can be solved by backward reasoning, as in ordinary logic programming. But, unlike ordinary logic programming, instead of failing if a sub-goal cannot be reduced to further sub-goals, if a sub-goal is an action, then an attempt can be made to make it true by executing it successfully. The result of the attempt is added to the knowledge base.

Conditional integrity constraints look like clauses in logic programs, but have a more general syntax[12]. To understand the difference between clauses and integrity constraints, it is useful to consider more closely the difference between data and integrity constraints (Nicolas and Gallaire, 1978).

In conventional relational databases, data is defined by relationships, which can be viewed in logical terms as facts (variable-free atomic sentences). For example:

The bus leaves at time 9:00.
The bus leaves at time 10:00. etc.

However, in deductive databases[13], which are a database-oriented variant of logic programming, relationships can also be defined by clauses. For example:

The bus leaves at time X:00 if X is an integer and $9 \leq X \leq 18$.

Given appropriate definitions for the predicates in its conditions, the clause replaces the 10 separate facts that would be needed in a conventional relational database.

Compare this clause with the conditional sentence:

If the bus leaves at time X:00,
then for some integer Y, the bus arrives at its destination at time X:Y & $20 \leq Y \leq 30$.

The existential quantifier in the conclusion of the sentence means that the sentence cannot be used to define any data, but can only be used as an integrity constraint. In a passive database, it would be used to reject any update that records an arrival time earlier than 20 minutes or later than 30 minutes after departure. However, in an active database, it could attempt to make its conclusion true and self-update the database with a record of the arrival.

---

[12] Integrity constraints can be arbitrary sentences of first-order logic. In fact, as we will see later when we discuss abductive logic programming, integrity constraints have the same syntax and semantics as goals. The difference between goals and integrity constraints is mainly terminological. Goals are one-off problems that need to be solved, including states of the world that an agent needs to achieve. Integrity constraints are persistent goals that need to be maintained over all states of the world.

[13] Deductive databases have the same syntax as logic programs, but are understood completely declaratively.

Obviously, the capability to make such an update lies outside the powers of even an active database. However, it is only a minor extension of a normal production system. In production systems, actions can be executed as soon as all of their conditions are satisfied. In this minor extension, actions have associated times, and they can be executed whenever their associated times are compatible with the actual time.

In addition to the pragmatic distinction between the use of clauses to define data and of integrity constraints to monitor data, there is a further distinction in the way that clauses and constraints are used for reasoning. In logic programming, clauses are used backwards to reduce goals to sub-goals. In deductive databases, integrity constraints are used forwards to derive consequences of updates (Kowalski et al, 1987). Forward reasoning with integrity constraints behaves like forward chaining to execute stimulus-response associations. This combination of backward reasoning with clauses and forward reasoning with integrity constraints facilitates the modelling of agents that combine the ability to plan pro-actively with the ability to behave reactively.

Intelligent Agents and AgentSpeak

The ability to combine planning for a future state of the world with reacting to the present state is a characteristic of both human and artificial agents. In the next section, we will see how these abilities are combined in abductive logic programming agents (ALP agents). ALP agents can be viewed in BDI (Belief, Desire, Intention) (Bratman, Israel, and Pollack,1988) terms, as agents whose beliefs are represented by clauses, desires (or goals) are represented by integrity constraints, and intentions are collections of actions with associated times to be performed in the future.

Arguably, the most influential of the BDI agent models, is the Procedural Reasoning System (Ingrand, George, and Rao, 1992), its successor dMARS (d'Inverno, 1998), and their abstraction and simplification AgentSpeak (Rao, 1996). Although the earliest of these systems were specified in multi-modal logics, their procedural implementations could not easily be related to their logical specifications. AgentSpeak abandoned the attempt to relate the implementations to their specifications, observing instead that "…one can view agent programs as multi-threaded interruptible logic programming clauses" (Rao, 1996).

However, the relationship between AgentSpeak and logic programming is restricted to the procedural interpretation of clauses. In fact, AgentSpeak is much closer to production systems than to logic programming. Like production systems, AgentSpeak programs have both a declarative and a procedural component, and the procedural component is an extended kind of production rule that does not have a logical semantics. The declarative component contains both *belief literals* (atoms and negations of atoms) and *goal atoms*, whereas the procedural component contains *plans* of the syntactic form:

Event E: conditions C $\Leftarrow$ *goals G and actions A.*

Here the event *E* can be the addition or the deletion of a belief or the addition of a goal. Plans are embedded in a cycle similar to the production system cycle. Production systems are the special case where the set of goals *G* is empty. Like production rules, plans are executed in the direction in which they are written, by a kind of forward chaining.

Here are some typical AgentSpeak plans:

+!quench_thirst: have_glass ⇐
!have_soft_drink;
fill_glass, drink.

+!have_soft_drink: soft_drink_in_fridge ⇐
open_fridge;
get_soft_drink.

+rock_seen(R): not battery_low ⇐
?location(R,L);
!move_to(L);
pick_up(R).

+ there is a fire: true ⇐ +there is an emergency.

Observations and actions do not have associated times, and the declarative memory provides only a snapshot of the current state of the world. To compensate for this lack of a temporal representation, the prefixes *+,-, !,* and *?* are used to stand for *add, delete, achieve,* and *test* respectively.

Like rules in production systems, plans in AgentSpeak do not have a logical semantics. However, in the special case where the triggering event *E* is the addition of a goal, and the plan has the form:

Goal E: conditions C ⇐ goals G and actions A.

the plan can be reformulated as a corresponding logic programming clause:

E' if C' and G' and A'and temporal constraints.

where the prefixed predicates of AgentSpeak are reformulated with explicit associated times. The corresponding clause is identical in behaviour to the plan, but also has a logical semantics. Using an (overly) simple, explicit representation of time[14], the clauses corresponding to first two plans illustrated above are:

quench_thirst at time T_5 if have_glass at time T_1 and
have_soft_drink at time T_2 and
fill_glass at time T_3 and drink at time T_4 and $T_1<T_2<T_3<T_4<T_5$.

have_soft_drink at time T_4 if soft_drink_in_fridge at time T_1 and
open_fridge at time T_2 and
get_soft_drink at time T_3 and $T_1<T_2<T_3<T_4$.

As in the case of stimulus-response production rules, the case where the triggering event is the addition or deletion of a belief can typically be represented by means of an integrity constraint. For example, the integrity constraint corresponding to the third plan above is:

[14] Notice that the explicit representation of time draws attention to implicit assumptions of persistence, for example that *have_glass at time T_1* persists until time T_5.

If rock_seen(R) at time T and not battery_low at time T and location(R,L) at time T then
move_to(L) at time T+1 and
pick_up(R) at time T+2.

In addition to goal-reduction rules and stimulus-response associations, there is a third kind of production rule and AgentSpeak plan, illustrated by the fourth plan above, in which the event *E* is an observation and the goals and actions are simply the addition of beliefs. This corresponds to a logic programming clause:

there is an emergency if there is a fire.

that is used to reason forwards in abductive logic programming, rather than backwards in normal logic programming, as we shall see again below.

Abductive logic programming (ALP)

Abductive logic programming (Kakas et al, 1998) is an extension of logic programming, which combines predicates that are defined by ordinary logic programs with abducible (undefined) predicates that are constrained by integrity constraints. In theory, integrity constraints can be expressed in full first-order logic, like goals in logic programs. In practice, however, syntactic restrictions are often imposed for the sake of efficiency or for naturalness of expression. The two most common restrictions limit the syntax of integrity constraints to the form of denials or the form of conditionals.

Consider the following abductive logic program (with minimal attention to temporal considerations), in which the integrity constraint is a denial expressed in the form of a conditional:

Program: *Grass is wet if it rained.*
Grass is wet if the sprinkler was on.

Abducible predicates: *it rained, the sprinkler was on, the sun was shining.*

Integrity constraints: *If it rained and the sun was shining then false.*
(i.e. *not it rained and the sun was shining.*)

Observation: *Grass is wet.*
Explanation: *the sprinkler was on.*

In classical logic, the observation has no deductive consequences. However, using the closed world assumption and reasoning backwards, it is possible to derive two alternative explanations of the observation:

it rained or the sprinkler was on.

As in the case of negation as failure, this argument can be formalised in two ways, either using the completion of the program, in which case the abduction becomes an object-level deduction, or using the autoepistemic semantics, in which case the abduction becomes a metalevel deduction.

Suppose, we are given the additional observation:

Observation: *the sun was shining.*

The new observation triggers the integrity constraint, and forward reasoning derives the consequence:

If it rained then false. (i.e. *not it rained.*)

from which it follows that:

the sprinkler was on.

is the only acceptable explanation of the observation that the *Grass is wet.*

This last inference can also be formalised in both the object language and in the metalanguage.

Now suppose that, instead of the additional observation, we have instead information about the relative probabilities of the abducible predicates, e.g.:

it rained	*(with probability .1)*
the sun was shining	*(with probability .5)*
the sprinkler was on	*(with probability .05)*
the sprinkler was off	*(with probability .95)*

Then it is possible to derive, in the obvious way, the associated probabilities of the two explanations, with the hypothesis that *it rained* being the more probable explanation.

ALP agents

In ALP agents (Kowalski and Sadri,1999; Kowalski 2001, 2006)), abductive logic programming is embedded in an observation-thought-decision-action cycle, which is similar to the production system and AgentSpeak cycles. The agent's beliefs are represented by clauses, goals by integrity constraints, and observations and actions by abducible predicates.

In the ALP agent cycle, reasoning can be interrupted, as in AgentSpeak, both by incoming observations and by outgoing actions. An incoming observation, for example, might trigger an integrity constraint and derive an action that needs to be performed immediately, interrupting the derivation of plans and execution of actions for goals that need to be achieved in the longer-term future. I general, it may be necessary to interleave actions that cannot be delayed with actions and plans for the future.

Consider the English sentences in the following example:

If you have an essay to write, then you study late in the library.
If there is an emergency, then get help.
If you press the alarm signal button, then you get help.
If there is a fire, then there is an emergency.

The four sentences all have conditional form. However, for an ALP agent, the first two sentences would function as *maintenance goals* and would be represented by integrity constraints. The other two sentences would function as beliefs and would be represented by clauses. The predicates

"there is a fire" and *"you press the alarm signal button"* are abducible predicates, representing possible observations and actions respectively. The predicate *you study late in the library* would be defined by other clauses.

All of the predicates, to be precise, would need to include associated times. These times would reflect the fact that getting help in an emergency is more urgent than studying in the library to write an essay.

Given an update recording an observation that *you have an essay to write*, forward reasoning with the first sentence would derive the achievement goal *you study late in the library* and set in train a process of goal-reduction and action execution to solve the achievement goal.

A second update recording an observation that *there is a fire* would interrupt this process and initiate a chain of forward reasoning to determine the consequences of the new update. Whereas in ordinary logic programming clauses can only be used backwards, in ALP clauses can be used backwards or forwards, depending on the circumstances. In particular, the fourth sentence above can be used forwards to classify the observation of fire as an emergency. (In other circumstances, it could also be used backwards to determine whether there is an emergency by checking whether there is a fire, or backwards to create an emergency by starting a fire!)

Forward reasoning with the fourth sentence derives the consequence *there is an emergency*. Forward reasoning with the second sentence derives the new achievement goal *you get help.* Backward reasoning with the second sentence reduces this goal to the abducible sub-goal, *you press the alarm signal button*. Because the sub-goal is an action abducible, it can be solved only by making it true, executing it successfully in the environment. Because the associated time of the action marks it as urgent, it would interrupt the process of studying late in the library, which could be returned to later.

In production systems and AgentSpeak, all four sentences would be represented uniformly as production rules or plans, and they would all be executed by forward chaining. Different uses of a belief would be represented by different rules or plans. For example, in AgentSpeak, the four sentences above would be represented by such plans as:

+ you have an essay to write: true ⇐ !you study late in the library
+there is an emergency: true ⇐ !get help.
+!get help: true ⇐ press the alarm signal button.
+ ?there is a fire: there is an emergency ⇐ true.
+ there is a fire: true ⇐ +there is an emergency.
+! there is an emergency: true ⇐ ! there is a fire.

It seems that this purely procedural, non-declarative representation is easier for many programmers to use than the combined procedural-declarative representation style of logic programming. Perhaps this is a symptom of the general confusion that is associated with the use of conditionals in natural language and of the fact that reasoning can be properly understood only in the context of its use by an agent to regulate its behaviour in the changing environment.

Conflict resolution in ALP agents – another role for probability

There is a growing view in Artificial Intelligence that the only kind of intelligence that makes sense is the intelligence needed by agents to maintain their well-being in co-operation and

competition with other agents in their environment. The presence of other agents and the uncertainty of the changing environment complicate the conflict resolution problem of deciding what to do when there are several alternatives to chose from.

Alternative ways of solving a goal may solve the goal with different degrees of utility and may have other desirable or undesirable consequences. Moreover, the consequences may depend upon the uncertain behaviour of other agents and of the environment. Ideally, an agent should choose an alternative that optimises its expected overall utility, including as many of its consequences as possible, as well as the overhead of performing the optimisation itself. However, it is often possible to use an inexpensive, fixed priority ordering of goals and beliefs to approximate the normatively ideal decision-theoretic analysis.

Consider, for example, the following two beliefs about alternative ways of going home for the weekend:

> *you will go home for the weekend if you have the bus fare and you catch a bus home.*
> *you will go home for the weekend if you have a car and you drive home.*

Both alternatives can be used to solve the goal of going home, but may do so with different utilities and probabilities. Taking into account the state of the car and the reliability of the bus service, it may be possible to associate explicit probabilities with the two clauses. For example:

> *you will go home for the weekend if you have the bus fare and you catch a bus home.*
> *(with probability .95)*
>
> *you will go home for the weekend if you have a car and you drive home.*
> *(with probability .80)*

However, as Poole (1993, 1997) has shown, the same effect can be achieved by associating probabilities with abducible predicates:

> *you will go home for the weekend if you have the bus fare and you catch a bus home*
> *and nothing goes wrong with the bus.*
> *you will go home for the weekend if you have a car and you drive home*
> *and nothing goes wrong with the car.*
> *nothing goes wrong with the bus. (with probability .95)*
> *nothing goes wrong with the car. (with probability .80)*

However, to do a full-scale decision-theoretic analysis, even if it were possible to determine the probabilities, it would still be necessary to determine the degree to which the alternatives accomplish their intended goal, as well as to the quantify the costs and benefits of other possible consequences. For example, driving the car may be more comfortable than taking the bus, and the bus schedule may be inconvenient. But taking the bus may cost less money, and, by contributing to the reduction of global carbon emissions, help to save the planet.

You could do a PhD on the problems of calculating the probabilities and utilities and on combining them to find the optimal solution. But, the result might be equivalent to just giving preference to the use of one clause over the other. This could be done, for example, with negation as failure, using the clauses Prolog fashion in a fixed order. For example:

> *you will go home for the weekend if you have the bus fare*

and you catch a bus home
and not something goes wrong with the bus.

you will go home for the weekend if you have a car
and you drive home
and not something goes wrong with the car.

Some conclusions

Artificial Intelligence contains a wealth of practical applications of different knowledge representation formalisms and their associated problem solving mechanisms, as well as theoretical results about the relationships between them. Conditionals in one form or another are a significant feature of many of these formalisms.

Although Artificial Intelligence has a different focus than Cognitive Psychology, it has borrowed many of its tools and techniques from the same background. It seems plausible, therefore, that some of its practical and theoretical results may also be relevant and useful in Cognitive Psychology. Conversely, psychological studies of human reasoning may be helpful in understanding why certain knowledge representation formalisms in Artificial Intelligence seem to be easier for some people to understand and to learn than other formalisms that are formally equivalent or even more powerful.

Production systems and logic programming are an obvious case, where there are well-established formal relationships, but an enormous psychological gap. Similar relationships hold between AgentSpeak and ALP agents, with a corresponding psychological chasm between them. Psychological studies of these gaps might be useful in helping to turn the formal results into practical applications.

The relationship between Bayesian networks and abductive logic programming is possibly an even more obvious case, where similar theoretic results are accompanied by a psychological chasm. This case is potentially of greater interest to cognitive psychologists, because it suggests that the psychological results arguing in favour of a probabilistic interpretation of human reasoning might be compatible with a logical interpretation.

References

Anderson, J. and Bower, G. (1973). *Human Associative Memory*. Washington, D.C.: Winston.

Bondarenko, A., Dung, P. M., Kowalski, R., and Toni, F. (1997) An Abstract Argumentation-theoretic Approach to Default Reasoning. *Journal of Artificial Intelligence 93 (1-2), 1997*, 63-101.

M. E. Bratman, D. J. Israel, and M. E. Pollack (1988) Plans and resource-bounded practical reasoning, *Computational Intelligence, vol. 4,* 349–355.

Clark, K.L. (1978) Negation by failure. In Gallaire, H. and Minker, J. [eds], *Logic and Databases*, Plenum Press, 293-322.

Colmerauer, A., Kanoui, H., Pasero, R. and Roussel, P. (1973) *Un systeme de communication homme-machine en Francais.* Research report, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II, Luminy.

Dancygier, B. (1998) *Conditionals and Prediction. Time, Knowledge, and Causation in Conditional Constructions*. Cambridge: Cambridge University Press.

Dung, P. M., Kowalski, R., and Toni, F. (2006) Dialectic proof procedures for assumption-based, admissible argumentation. *Journal of Artificial Intelligence 170(2), 2006*,114-159.

Ingrand, F. F. , Georgeff, M. P. and Rao, A. S. (1992) An architecture for real-time reasoning and system control. *IEEE Expert, 7(6).*

d'Inverno, Luck, M., M., Georgeff, M. P., Kinny, D., and Wooldridge, M. (1998) A Formal Specification of dMARS. In: *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages. Lecture Notes in Artificial Intelligence, 1365.* Springer-Verlag, 155-176.

Fung, T.H. and Kowalski, R. (1997) The IFF Proof Procedure for Abductive Logic Programming. *Journal of Logic Programming.*

Gelfond, M. and Lifschitz, V. (1990) Logic programs with classical negation. *Proceedings of the Seventh International Conference on Logic Programming,* MIT Press, 579-597.

Kakas, T., Kowalski, R., Toni, F. (1998) The Role of Logic Programming in Abduction. In: Gabbay, D., Hogger, C.J., Robinson, J.A. (eds.*): Handbook of Logic in Artificial Intelligence and Programming 5*. Oxford University Press 235-324.

Kowalski, R., (1974) Predicate Logic as Programming Language. In *Proceedings IFIP Congress*, Stockholm, North Holland Publishing Co., 569-574. Reprinted In *Computers for Artificial Intelligence Applications*, (eds. Wah, B. and Li, G.-J.), IEEE Computer Society Press, Los Angeles, 1986, 68-73.

Kowalski, R. (1979) *Logic for Problem Solving.* North Holland Elsevier.

Kowalski, R. (2001) Artificial intelligence and the natural world. *Cognitive Processing, 4,* 547-573.

Kowalski, R. (2006) The Logical Way to be Artificially Intelligent. *Proceedings of CLIMA VI* (eds. F. Toni and P. Torroni) Springer Verlag, LNAI, 1-22.

Kowalski, R., Sadri, F. and Soper, P. (1987) Integrity Checking in Deductive Databases. In: *Proceedings of VLDB,* Morgan Kaufmann, Los Altos, Ca. 61-69.

Kowalski, R., Sadri, F. (1999) From Logic Programming towards Multi-agent Systems. *Annals of Mathematics and Artificial Intelligence.* Vol. 25 391-419.

Newell, A. (1973). Production Systems: Models of Control Structure. In W. Chase (ed): *Visual Information Processing* 463-526 New York: Academic Press 463-526.

Nicolas, J.M., Gallaire, H. (1978) Database: Theory vs. interpretation. In Gallaire, H., Minker, J. (eds.*): Logic and Databases.* Plenum, New York.

Pearl, J. 1988: *Probabilistic Reasoning in Intelligent Systems*. San Mateo, CA: Morgan Kaufmann.

Politzer G. and Bonnefon J-F. (2006) Two varieties of conditionals and two kinds of defeaters help reveal two fundamental types of reasoning. *Mind and Language,***21**, 484-503.

Politzer G. and Bonnefon J-F. (this volume)

Poole, D., Goebel, R. and Aleliunas R. (1987) Theorist: a logical reasoning system for defaults and diagnosis. In N. Cercone and G. McCalla (Eds.) *The Knowledge Frontier: Essays in the Representation of Knowledge*, Springer Verlag, New York, 1987, 331-352.

Poole, D. (1993) Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1) 81–129.

Poole, D. (1997) The independent choice logic for modeling multiple agents under uncertainty. *Artificial Intelligence.* Vol. 94 7-56

Rao, A.S. (1996) Agents Breaking Away, *Lecture Notes in Artificial Intelligence, Volume 1038,* (eds Walter Van de Velde and John W. Perrame) Springer Verlag, Amsterdam, Netherlands.

Reiter, R. (1978) On closed world data bases. In H. Gallaire and J. Minker, editors: *Logic and Data Bases*, Plenum, New York. 119-140.

Shortliffe, 1974

Simon, H. (1999) Production Systems. In Wilson, R. and Keil, F. (eds.): *The MIT Encyclopedia of the Cognitive Sciences.* The MIT Press. 676-677.

Stenning, K. and van Lambalgen M., (2008) *Human reasoning and cognitive science.* MIT Press. (in press).

Stenning, K. & van Lambalgen, M. (2004) A little logic goes a long way: basing experiment on semantic theory in the cognitive science of conditional reasoning. *Cognitive Science*, 28:4, 481-529.

Stenning, K. & van Lambalgen, M. (2005) Semantic interpretation as computation in nonmonotonic logic: the real meaning of the suppression task. *Cognitive Science*, 29(6), 919–960.

Thagard, P. (1996) *Mind: Introduction to Cognitive Science.* MIT Press.