Reasoning with Conditionals in Artificial Intelligence

Robert Kowalski
Department of Computing
Imperial College London

January 2008 (revised December 2009)

1 Introduction

Conditionals of one kind or another are the dominant form of knowledge representation in Artificial Intelligence. However, despite the fact that they play a key role in such different formalisms as production systems and logic programming, there has been little effort made to study the relationships between these different kinds of conditionals. In this paper I present a framework that attempts to unify the two kinds of conditionals and discuss its potential application to the modelling of human reasoning.

Among the symbolic approaches developed in Artificial Intelligence, production systems have had the most application to the modelling of human thinking. Production systems became prominent in the late 1960s as a technology for implementing expert systems in Artificial Intelligence, and then in the 1970s as a cognitive architecture in Cognitive Psychology. However, although they have been used extensively to study human skills, they seem to have had little impact on studies of human reasoning. This lack of interaction between cognitive architectures based on production systems and studies of human reasoning is especially surprising because both emphasise the central role of conditionals.

Conditionals in production systems (called production rules) have the form *if conditions then actions* and look like logical implications. Indeed, some authors, such as Russell and Norvig (2003), take the view that production rules are just logical implications used to reason forward to derive candidate *actions* from *conditions* (*modus ponens*). However, other authors, such as Thagard (2005) deny any relationship between logic and production rules at all.

In some characterisations of production rules, they are described as expressing procedural knowledge about what to do when. In this paper, I will argue that production rules can be given a logical semantics, and this semantics can help to clarify their relevance for modelling human reasoning.

Conditionals are also a distinctive feature of logic programming, which has been used widely, also since the 1970s, both to implement practical applications and to formalise knowledge representation in Artificial Intelligence. Conditionals in logic programming have both a logical interpretation as conditionals of the form *conclusion if conditions* and an interpretation as procedures that reduce the goal of establishing the *conclusion* to the sub-goals of establishing the *conditions.* Goal-reduction is a form of *backward reasoning.* Although forward reasoning has been studied extensively in psychology, backward reasoning seems to have received little attention in studies of human reasoning.

Instead, studies of reasoning in Psychology emphasize the use of classical negation, including its use in such classically valid inferences as *modus tollens* and in such classical fallacies as denial of the antecedent. In contrast, classical negation has had relatively few applications in Artificial Intelligence, where negation as failure and the closed world assumption (*not P* holds if and only if *P* does not hold) have been more widely used in practice.

Studies of reasoning in Psychology treat affirmation of the consequent as a fallacy. In Artificial Intelligence, however, it is treated as abductive inference. Both abduction and negation as failure are forms of non-monotonic reasoning.

In this paper, I will outline an abductive logic programming (ALP) approach that aims to reconcile production rules and logic within a unifying agent-based framework. In this framework, logic programs are conditionals used to represent an agent's beliefs, and have a descriptive character. Production rules are conditionals in logical form, used to represent the agent's goals, and have deontic force. Abduction is used both to represent hypotheses that explain the agent's observations and actions that might achieve the agent's goals. Abductive hypotheses can have associated probabilities.

The ALP agent framework embeds goals and beliefs in an observation-thought-decision-action cycle, similar to the production system cycle, and similar to other agent cycles developed in Artificial Intelligence. In the ALP agent framework, the thinking component of the cycle is a combination of forwards reasoning, as in production systems, and backwards reasoning, as in logic programming. Negation is represented both by means of negation as failure and by means of integrity constraints.

Because this book is directed primarily towards a psychological audience and because my own expertise lies in Artificial Intelligence, I will focus on the potential contribution of Artificial Intelligence to studies of human reasoning. However, I anticipate that there is much to be gained from applying results about human reasoning to Artificial Intelligence. In particular, it seems likely that results about the psychology of reasoning with negation can suggest useful directions in which to extend AI reasoning techniques.

The remainder of the paper has four main parts, dealing with production systems, logic programs, the relationship between production systems and logic programs, and the ALP agent model. In addition, there is a section on integrity constraints, to help motivate the interpretation in ALP of stimulus-response production rules as a species of integrity constraint, and a section on the agent language AgentSpeak, which can be regarded as an extension of production systems in which actions are generalised to plans of action.

The semantics and proof procedures of logic programming and ALP are based on classical logic, but are sufficiently different to be a potential source of difficulty for some readers. For this reason, I have put details about the minimal model semantics in an appendix. I have also included an appendix about the interpretation of natural language conditionals in logic programming terms, because it interrupts the main flow of the paper.

Before settling down to the main part of the paper, I first present a motivating example, discuss the Wason selection task, and present some historical background.

2 A motivating example

Consider the following real world example of a sign posted in carriages on the London underground:

Emergencies:
Press the alarm signal button to alert the driver.
The driver will stop if any part of the train is in a station.
If not, the train will continue to the next station, where help can more easily be given.
There is a 50 pound penalty for improper use.

The sign is intended to be read and for its contents to be assimilated by passengers traveling on the underground. For this purpose, a passenger needs to translate the English text from natural language into a mental representation. In AI, such mental representations include logic, logic programs, production rules and other procedural representations. Here is the main part of a translation of the London underground sign into the ALP agent representation that I will describe later in the paper:

> *You alert the driver that there is an emergency*
> *if you are on the underground and there is an emergency*
> *and you press the alarm signal button.*
> *The driver will stop the train immediately if you press the alarm*
> *signal button and any part of the train is in a station.*
> *The driver will stop the train (and help can more easily be given)*
> *at the next station if you press the alarm signal button*
> *and not any part of the train is in a station.*
> *You will be liable to a 50 pound penalty*
> *if you press the alarm signal button improperly.*

The first sentence of the translation shows how procedures are represented in logic programming form as conditionals that are used backwards to reduce goals that match the conclusion of the conditional to sub-goals that correspond to the conditions. The first, second and third sentences show that the conditional form of natural language sentences should not to be taken literally, but may need to include extra conditions (such as *you press the alarm signal button*) and extra qualifications (such as the driver will stop *the train*) implicit in the context. The fourth sentence shows that the meaning of a natural language sentence may be a conditional whether or not the surface structure of the sentence contains an explicit mark of a conditional.

However, the translation of the sign does not represent its full import, which depends upon the reader's additional background goals and beliefs. These might include such goals and beliefs as[1]:

If there is an emergency then you get help.
You get help if you are on the underground and you alert the driver.

I will argue later that the first sentence is a maintenance goal, which is a special case of an integrity constraint of the kind used to maintain integrity in active database systems. Maintenance goals are similar to plans in the intelligent agent programming language AgentSpeak. I will also argue that they generalize production rules in production systems.

[1] These two sentences also help to explain the significance of the phrase *where help can more easily be given*. The passenger's background knowledge may also contain other ways of getting help. Arguably, the phrase is intended to help the passenger to evaluate alternative candidate actions, in deciding what to.

In the ALP agent framework, goals and beliefs are embedded in an observation-thought-decision-action cycle. Given an observation of an emergency, the maintenance goal would be used to reason forward, to derive the achievement goal of getting help. The logic program would be used to reason backward, to reduce the goal of getting help to the action sub-goal of pressing the alarm signal button. Assuming that there are no other candidate actions to consider, the agent would commit to the action and attempt to solve it by executing it in the environment.

3 The Selection Task

I will not attempt to analyse the Wason selection task and its variants (Wason, 1968) in detail, but will outline instead how the ALP agent framework addresses some of the problems that have been observed with human performance in psychological experiments.

Consider the following simplified abstraction of the selection task presented to a subject in natural language:

> *Given some incomplete information/observations,*
> *what conclusions can be derived using the conditional if P then Q?*

The task is given with a concrete instance of the conditional, such as *if there is a vowel on one side of a card then there is an even number on the other side* and *if a person is drinking alcohol in a bar then the person is over eighteen years old*. To solve the task, the subject first needs to translate the conditional into a mental representation. The subject then needs to use the mental representation of the conditional, possibly together with other background goals and beliefs, to derive conclusions from the observations. I will argue that many of the reasons for the variation in subjects' performance on the selection task can be demonstrated using the ALP agent model with this simplified formulation of the task.

In the ALP agent model, the subject needs first to decide whether the English language conditional should be understood as a belief or as a goal. If it is understood as a belief, then it is represented as a clause in a logic program. If it is understood as a goal, then it is represented as an integrity constraint, to monitor updates. These two alternatives, understanding the conditional as a belief or as a goal, correspond roughly to the descriptive and deontic interpretations of the conditional respectively. In concrete instances of the task, the syntactic form of the conditional, its semantic content, and the way in which the task is formulated can all influence the interpretation.

Suppose the subject interprets the conditional as a belief, which is then represented as a logic programming clause:

> *Q if P*

Assuming that there are no other clauses, including background clauses, that have the same conclusion *Q*, the clause is then interpreted as the *only* way of concluding *Q*. In the completion semantics (Clark, 1978) of logic programming, this can be expressed explicitly in the form:

> *Q if and only if P*

The equivalence here is asymmetric in the sense that the formula on the right hand side is taken to be the definition of the predicate on the left hand side of the equivalence, and not the other way around.

In the ALP agent model described in this paper, given an observation *O*, the agent can reason both forwards from *O*, to derive consequences of the observation, and backwards from *O*, to derive explanations of the observation. In the selection task, therefore, forward reasoning can be used to derive *Q* from an observation of *P*, and backward reasoning can be used to derive *P* as an explanation of an observation of *Q*. These are the classic responses to Wason’s original card version of the selection task.

The modus tollens derivation of *not P* from *not Q* is also possible, but more difficult, because it is first necessary to derive *not Q* from some initial positive observation *Q’*. This is because, in general, an agent’s observations are represented only by atomic sentences. Negations of atomic sentences need to be derived from positive observations. In card versions of the selection task, for example, the conclusion that a card does not have a vowel on one of its sides needs to be derived by a chain of reasoning, say from an observation that it has the letter B on that side, to the conclusion that it has a consonant on that side, to the conclusion that it does not have a vowel on that side. As Sperber, Cara, and Girotto (1995) argue, the longer the derivation, and the greater the number of irrelevant, alternative derivations, the less likely it is that a subject will be able to perform the derivation.

In the ALP agent model, the relationship between positive and negative concepts, needed to derive a negative conclusion from a positive observation, is expressed in the form of integrity constraints, such as:

if mortal and immortal then false	i.e.	*not(mortal and immortal)*
if odd and even then false	i.e.	*not(odd and even)*
if vowel and consonant then false	i.e.	*not(vowel and consonant)*
if adult and minor then false	i.e.	*not(adult and minor)*
etc.		

and more generally as:

if Q’ and Q then false	i.e.	*not(Q’ and Q)*

where *Q’* is the positive observation or some generalization of the observation (such as the card has a consonant on the face) and *if Q then false* is an alternative syntax for the negation *not Q*.

In the ALP agent model, given a concrete observation that leads by a chain of forward reasoning to the conclusion *Q’*, a further step of forward reasoning with the integrity constraint is needed to derive *not Q*. Backward reasoning with the conditional (replacing *Q* by its definition *P*) then derives *not P*.

Thus in card versions of the Wason selection task, when the conditional is interpreted as a belief, the derivations of *P* from *Q* and of *Q* from *P* are straight-forward, the derivation of *not P* from some contrary positive observation that implies *not Q* is difficult, but possible. But the derivation of *not Q* from *not P* is not possible at all, because of the asymmetry of the equivalence *Q if and only if P*.

Suppose instead that the subject interprets the conditional as a goal. In the ALP agent model, this is represented by an integrity constraint of the form:

if P then Q.

Such integrity constraints are used to reason forwards, like production rules, in this case to derive *Q* from *P*. They are not used backwards to derive *P* from *Q*, which is consistent with experimental data for deontic versions of the selection task.

As in the case where the conditional is interpreted as a belief, negative premises, such as *not P* and *not Q*, need to be derived by forward reasoning from positive atomic observations, using integrity constraints of such form as:

if Q' and Q then false
if P' and P then false

and the same considerations of computational complexity apply.

Assuming that the positive observations imply both *Q'* and *P'* and that both:

if Q then false
if P then false

have been derived, then the only further inference that is possible is:

From	*if P then Q*
and	*if Q then false*
derive	*if P then false*
i.e.	*not P.*

However, this inference step, which is both easy and natural for human subjects, is not possible in some existing ALP proof procedures. I will argue later that this is because these proof procedures implement the wrong semantics for integrity constraints. The semantics needed to justify this inference step is the *consistency view of integrity constraint satisfaction*. I will discuss this problem and its solution again in section 10.2. In the meanwhile, it is interesting to note that the selection task may suggest a direction for solving technical problems associated with proof procedures developed for practical applications in AI.

4 A personal view of the history of logic in AI

To put the issues dealt with in this paper into context, it may be helpful to consider their historical background.

4.1 Heuristic versus formal approaches

It is widely agreed that the most important work concerning logic in the early days of AI was the Logic Theorist (Newell et al, 1957), which managed to prove 38 of the first 52 theorems in Chapter 2 of Principia Mathematica. The Logic Theorist pioneered the heuristic approach, employing three inference rules, “backward reasoning”, “forward chaining” and “backward chaining”, without traditional logical concerns about semantics and completeness. In contrast with formal theorem-proving techniques, these heuristic inference rules behaved naturally and efficiently.

Logic Theorist led to GPS, a general problem solver, not directly associated with logic, and later to production systems. Production rules in production systems resembled conditionals in traditional logic, but did not suffer from the inscrutability and inefficiencies associated with formal theorem-proving.

In the meanwhile, McCarthy (1958) advocated a formal approach, developing the Advice Taker, using formal logic for knowledge representation and using theorem-proving for problem-solving. In the Advice-Taker approach, the theorem-prover was a "black box", and there was no attempt to relate the behaviour of the theorem-prover to common sense techniques of human problem-solving. The theorem-proving approach led to the development of question-answering systems, using complete and correct theorem-provers, based on mathematical logic.

Resolution (Robinson, 1965) was developed within the formal logic tradition, with mainly mathematical applications in mind. Its great virtue was its simplicity and uniformity, compressing the rules and logical axioms of symbolic logic into a single inference rule. It could not be easily understood in human-oriented terms, but was presented as a machine-oriented logic. Its early versions were very inefficient, partly because the resolution rule did not have an intuitive interpretation.

Towards the end of the 60s, there were two main trends among symbolic approaches to AI: the heuristic (sometimes called "scruffy" or "strong") approach, mainly associated with production systems, which behaved in human-intelligible terms, uninhibited by mathematical concerns about semantics, but emphasizing the importance of domain-specific knowledge; and the formal ("neat" or "weak") approach, exemplified by resolution-based systems, emphasizing domain-independent, general-purpose problem-solving. Production systems were beginning to have applications in expert systems, and resolution was beginning to have applications in mathematics and question-answering.

4.2 Procedural representations of knowledge and logic programming

In the meanwhile, critics of the formal approach, based mainly at MIT, began to advocate procedural representations of knowledge, as superior to declarative, logic-based representations. This led to the development of the knowledge representation and problem-solving languages Planner and micro-Planner. Winograd's PhD thesis (1971), using micro-Planner to implement a natural language dialogue for a simple blocks world, was a major milestone of this approach. Research in automated theorem-proving, mainly based on resolution, went into sharp decline.

The battlefield between the logic-based and procedural approaches moved briefly to Edinburgh during the summer of 1970 at one of the Machine Intelligence Workshops organized by Donald Michie (van Emden, 2006). At the workshop, Pappert and Sussman from MIT gave talks vigorously attacking the use logic in AI, but did not present a paper for the proceedings. This created turmoil among researchers in Edinburgh working in resolution theorem-proving. However, I was not convinced that the procedural approach was so different from the SL-resolution system I had been developing with Donald Kuehner (1971).

During the next couple of years, I tried to reimplement Winograd's system in resolution logic and collaborated on this with Alain Colmerauer in Marseille. This led to the procedural interpretation of Horn clauses (Kowalski 1973/1974) and to Colmerauer's development of the programming language Prolog. I also investigated other interpretations of resolution logic in problem solving terms (Kowalski 1974/1979), exploiting in particular the fact that all sentences in resolution logic can be expressed in conditional form.

However, at the time, I was not aware of the significance of production systems, and I did not understand their relationship with logic and logic programming. Nor did I appreciate that logic programming would become associated with general-purpose problem-solving, and that this would be misunderstood as meaning that logic programming was suitable only for representing "weak" general-purpose knowledge. I will come back to this problem later in the paper.

4.3 Non-monotonic reasoning, abduction, and argumentation

In the mid-1970s, Marvin Minsky (1974) launched another attack against logic from MIT, proposing frames as a representation of stereotypes. In contrast with formal logic, frames focussed on the representation of default knowledge, without the need to specify exceptions strictly and precisely. This time the logic community in AI fought back with the development of various non-monotonic logics, including circumscription (McCarthy, 1980), default logic (Reiter, 1980), modal non-monotonic logic (McDermott and Doyle, 1980), autoepistemic logic (Moore, 1985) and negation as failure in logic programming (Clark, 1978).

(Poole et al, 1987) argued that default reasoning can be understood as a form of abductive reasoning. Building upon their results, Eshghi and I (1989) showed that negation as failure in logic programming can also be understood in abductive terms. More recently, Dung and his collaborators (Bondarenko et al 1997 ; Dung et al, 2006) have shown that most non-monotonic logics can be interpreted in terms of arguments and counter-arguments supported by abductive hypotheses. Moreover, Poole (1993, 1997) has shown that Baysian networks can be understood as abductive logic programs with assumptions that have associated probabilities.

4.4 Intelligent agents

Starting in the late 1980s and early 1990s, researchers working in the formal logic tradition began to embed logic-based representations and problem-solving systems in intelligent agent frameworks. The intelligent agent approach was given momentum by the textbook of Russell and Norvig (2003), whose first edition was published in 1995. In their book, Russell and Norvig credit Newell, Laird and Rosenblum (Newell, 1990; Laird *et al*., 1987) with developing SOAR as the "best-known example of a complete agent architecture in AI".

However, SOAR was based on production systems and did not have a logical basis. Among the earliest attempts to develop formal, logic-based agent architectures were those of Rao and Georgeff (1991) and Shoham (1991), both of which were formulated in BDI (Belief, Desire, Intention) terms (Bratman, Israel, and Pollack,1988) . Although their formal specifications were partly formulated in modal logics, their implementations were in procedural languages that looked like extensions of production systems. The ALP agent model (Kowalski and Sadri,1999; Kowalski 2001, 2006) was developed to reconcile the use of logic for agent specification with its use for implementation. Instead of using modalities to distinguish between beliefs and desires, it uses an extension of the database distinction between data and integrity constraints, treating beliefs like data and desires (or goals) like integrity constraints.

5 Production systems

Production systems were originally developed by the logician Emil Post as a mathematical model of computation, and later championed by Alan Newell as a model of human problem solving (Newell, 1973; Anderson and Bower, 1973). Production systems restrict knowledge representation to a combination of *facts*, which are atomic sentences, and *condition-action rules*, which have the syntax of conditionals, but arguably do not have a logical semantics. They also restrict reasoning to a kind of modus ponens, called *forward chaining.*

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

If conditions C, then actions A.

Condition-action rules are also called *production rules*, just plain *rules*, or *if-then rules.* The most typical 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 an 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, so that the first rule to be triggered is executed first. At the opposite extreme, actions can be chosen by means of a full scale decision-theoretic analysis, analysing 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 arguably have no declarative semantics at all. Production rules look like logical implications, without the semantics of implications. Moreover, forward chaining looks like *modus ponens*, but it is not truth-preserving, because there is no notion of what it means for a condition-action rule to be true.

The similarity of production rules to implications is a major source of confusion. Some authors (Thagard, 2005) maintain that production systems enable *backward reasoning*, from actions that are goals to conditions that are sub-goals. MYCIN (Shortliffe, 1976), 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 might be better understood as using an informal kind of logic.

When production rules are used to implement stimulus-response associations, they normally achieve unstated goals implicitly. Such implicit 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*, which is a sub-goal of *passing the course*, which is a sub-goal of *getting a degree*, which is a sub-goal of the top-level goal of *becoming a success in life*.

In contrast with systems that explicitly reduce goals to sub-goals, 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 emergent, rather than explicit.

Although the natural use of production systems is to implement stimulus-response associations, they are also used to simulate backward reasoning, not by executing production rules backwards, but by treating goals as facts and by forward chaining with rules of the form:

> *If goal G and conditions C then add H as 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 (2005) also draws attention to this use of production rules and claims that such rules cannot be represented in logical form. 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 motivates the terminology “conflict resolution” to describe the process of reconciling conflicting commands.

The differences between production systems and logic programming are of two kinds. There are technical differences, such as the fact that production rules are executed in the forward direction, the fact that conclusions of production rules are actions or recommendations for actions, and the fact that production rules are embedded in a cycle in which conflict resolution is performed to choose between candidate actions. And there are cultural differences, reflecting a more casual attitude to issues of correctness and completeness, as well as a greater emphasis on “strong”, domain-specific knowledge in contrast to “weak” general-purpose problem-solving methods.

The technical differences are significant and need to be treated seriously. In the ALP agent model, in particular, we take account of forward chaining by using logic programs and integrity constraints to reason forwards; we take account of the imperative nature of production rules by treating them as goals rather than as beliefs; and we take account of the production system cycle by generalising it to an agent cycle, in which thinking involves the use of both goals and beliefs.

However, the cultural differences are not technically significant and are based upon a failure to distinguish between *knowledge representation* and *problem-solving*. Knowledge can be weak or strong, depending on whether it is general knowledge that applies to a wide class of problems in a given domain, like axioms of a mathematical theory, or whether it is specialised knowledge that is tuned to typical problems that arise in the domain, like theorems of a mathematical theory.

However, some strong knowledge may not be derivable from general-purpose knowledge in the way that theorems are derivable from axioms. Strong knowledge might be more like a mathematical conjecture or a rule of thumb, which is incomplete and only approximately correct. Strong knowledge can exist in domains, such as expert systems, where there exists no weak knowledge that can serve as a general axiomatisation.

Logic is normally associated with weak knowledge, like mathematical axioms; and production systems are associated with strong knowledge, as in expert systems. But there is no reason, other than a cultural one, why logic can not also be used to represent strong knowledge, as in the case of mathematical theorems or conjectures.

In contrast with knowledge, which can be weak or strong, problem-solving methods are generally weak and general-purpose. Even production systems, with their focus on strong knowledge, employ a weak and general-purpose problem-solving method, in their use of forward chaining. Logic programs similarly employ a weak and general-purpose method, namely backward reasoning.

6 Logic programming

6.1 A short introduction to logic programming

Logic programming has three main classes of application: as a general-purpose programming language, a database language, and a knowledge representation language in AI. As a programming language, it can represent and compute any computable function. As a database language, it generalises relational databases, to include general clauses in addition to facts. And as a knowledge representation language it is a non-monotonic logic, which can be used for default reasoning. Its most well-known implementation is the programming language Prolog, which combines pure logic programming with a number of impure features.

In addition to the use of logic programming as a normative model of problem solving (Kowalski, 1974/79), Stenning and van Lambalgen (2004, 2005, 2008) have investigated its use as a descriptive model of human reasoning.

Logic programs (also called *normal logic programs*) are sets of conditionals of the form:

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. Conditionals in logic programs are also called *clauses*. *Horn clauses* [2] are the special case where all of the conditions are atomic formulae. *Facts* are the

[2] Horn clauses are named after the logician Alfred Horn, who studied some of their model-theoretic properties.

special case where *n* = 0 (there are no conditions) and there are no variables. Sometimes clauses that are not facts are also called *rules*, inviting confusion with production rules.

Goals (or *queries*) are conjunctions of literals, syntactically just like the conditions of clauses. However, all variables are implicitly existentially quantified, and the intention of the goal is to find an instantiation of the variables that makes the goal hold.

For example, the three sentences:

> *If you have the bus fare and you catch a bus*
> *and not something goes wrong with the bus journey,*
> *then you will go home for the weekend.*
> *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. Notice that the second sentence can be regarded as an imprecise version of the first sentence. Notice too that the first two clauses both express "strong" domain-specific knowledge, rather than the kind of weak knowledge that would be necessary for general-purpose planning.

The sentence *you will go home for the weekend* is a simple, atomic goal.

Backward reasoning (from conclusions to conditions) treats conditionals as goal-reduction procedures:

> *to show/solve H, show/solve* B_1 *and … and* B_n.

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 chaining" in the direction in which the conditional is written. The Prolog syntax for clauses:

> *H :-* B_1 *,… ,* B_n.

is deliberately ambiguous, so that clauses can be read either declaratively as conditionals written backwards or procedurally as goal-reduction procedures executed forwards.

Whereas positive, atomic goals and sub-goals are solved by backward reasoning, negative goals and sub-goals of the form *not G,* where *G* is an atomic sentence, are solved by *negation as failure: not G* succeeds if and only if backward reasoning with the sub-goal *G* does not succeed.

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

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

then the consequence:

> *The apple is red.*

follows as a goal, because there is no clause whose conclusion matches the sub-goal *the apple is illuminated by a red light*, and therefore the two conditions for the only clause that can be used in solving the goal both hold. However, given the additional clause *the apple is illuminated by a red light*, the sub-goal now succeeds and the top-level goal now fails, non-monotonically withdrawing the consequence *The apple is red.* However, the consequence *not the apple is red* now succeeds instead.

Goals and conditions of clauses can be generalised from conjunctions of literals to arbitrary formulae of first-order logic. The simplest way to do so is to use auxiliary predicates and clauses (Lloyd and Topor, 1984). For example, the goal:

> *Show that for all exams,*
> *if the exam is a final exam, then you can study for the exam in the library.*

can be transformed into the normal logic programming form:

> *Show that not the library is useless.*
> *the library is useless if the exam is a final exam and*
> *not you can study for the exam in the library.*

This is similar to the transformation[3] noted by Sperber, Cara, and Girotto (1995) needed to obtain classically correct reasoning with conditionals in variants of the Wason Selection Task. However, it is important to note that the transformation applies in logic programming only when the conditional is interpreted as a goal, and not when it is interpreted as a clause.

The computational advantage of the transformation is that it reduces the problem of determining whether an arbitrary sentence of first-order logic holds with respect to a given logic program to the two simple inference rules of backward reasoning and negation as failure alone.

6.2 Strong versus weak methods in logic programming

[3] The use of the auxiliary predicate is merely a technical device, which is useful for maintaining the simple syntax of goals and clauses. However, it is also possible to employ a more natural syntax in which the conditional is written directly in the form of a denial: *Show that not there is an exam, such that (the exam is a final exam and not you can study for the exam in the library).*

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 primarily on the declarative reading and are disappointed when their programs fail to run efficiently. Other programmers focus instead on the procedural reading and loose the benefits of the declarative reading.

Part of the problem is purely technical, because many declarative programs, like the clause:

> *Mary likes a person if the person likes Mary.*

reduce goals to similar sub-goals repeatedly without termination. This problem can be solved at least in part by employing more sophisticated (“stronger”), but still general-purpose problem-solvers that never try to solve similar sub-goals more than once (Sagonas et al, 1994).

However, part of the problem is also psychological and cultural, because many programmers use logic only to specify problem domains, and not to represent useful knowledge for efficient problem-solving in those domains.

The planning problem in AI is a typical example. In classical planning, the problem is specified by describing both an initial state and a goal state, and by specifying the preconditions and postconditions of atomic actions. To solve the problem, it is then necessary to find a sequence of atomic actions that transforms the initial state into the goal state. This is sometimes called *planning from first principles*, or by *brute force*. In many domains it is computationally explosive.

The alternative is to use a collection of precompiled plan schemata that apply to a wide class of commonly occurring problems in the given domain. These plan schemata typically reduce a top-level goal to high-level actions, rather than to atomic actions, in a hierarchy of reductions from higher-level to lower-level actions. This is sometimes called *planning from second principles*, and it can be very efficient, although it may not be as complete as planning from first principles.

Algorithms and their specifications are another example. For instance, the top-level specification of the problem of sorting a list can be expressed as:

> *list L’ is a sorted version of list L if L’ is a permutation of L and L’ is ordered.*

No “weak” general-purpose problem-solving method can execute this specification efficiently. The Prolog implementation of backward reasoning, in particular, treats the specification as a procedure:

> *To sort list L obtaining list L’, generate permutations L’ of L,*
> *until you find a permuation L’ that is ordered.*

However, there exist a great variety of efficient sorting algorithms, which can also be expressed in logic programming form. For example, the top-level of the recursive part of quicksort:

> *To quicksort a list L into L’, split L into two smaller lists L_1 & L_2 and*
> *quicksort L_1 into L'_1 and quicksort L_2 into L'_2 and shuffle L_1 & L_2 together into L’.*

I leave the Horn clause representation of the procedure to the reader. Suffice it to say that quicksort is a sufficiently “strong” problem-solving method that it behaves efficiently even when executed by a weak problem-solving method, such as backward reasoning. Stronger, but still general-purpose methods, such as parallelism, can improve efficiency even more.

6.3 Two kinds of semantics

There are two main kinds of semantics for logic programs, with two main ways of understanding what it means for a goal to hold. Both of these semantics interpret the clauses of a program as the *only* clauses that can be used to establish the conclusions of clauses. This interpretation is a form of the *closed world assumption* (Reiter, 1978).

In the *minimal model semantics*, a logic program defines a set of minimal models, and a goal holds if it is *true* in one or more of these models. The minimal model semantics comes in a *credulous* version, in which a goal holds if it is true in *some* minimal model; and a *sceptical* version, in which a goal holds if it is true in *all* minimal models.

The notion of minimal model is very simple in the case of Horn clause programs, because every Horn clause program has a unique minimal model, in which an atomic sentence is true if and only if it is true in all models. However, it is more complicated for normal logic programs with negative conditions. For this reason, a more detailed discussion of the minimal model semantics is given in the appendix.

The main (and arguably simpler) alternative to the minimal model semantics is the *completion semantics*, in which clauses are understood elliptically as the if-halves of definitions in if-and-only-if form (Clark, 1978). If a predicate does not occur as the conclusion of a clause, then it is deemed to be *false.* This interpretation is called the *predicate (or Clark) completion*. A goal holds in the completion semantics if it is a *theorem*, logically entailed by the completion of the logic program.

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[4] 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*.)

which logically entails the conclusion: *not you get help*.

Negation as *finite* failure is *sound* with respect to the completion semantics:

> *not G* is logically entailed by the completion
> if *G fails finitely*.

[4] The use of *if and only if* here does not make the two sides of the “equivalence” symmetric. The left-hand side is a predicate defined by the formula on the right-hand side.

It is also complete in many cases.

The completion and minimal model semantics have different notions of what it means for a goal, which in both cases can be any formula of first-order logic, to hold with respect to a given logic program. In both semantics, the same proof procedures of backward reasoning and negation as failure can be used. However, in the minimal model semantics, negation as failure includes possibly infinite failure, whereas in the completion semantics, negation as failure is finite.

Given a program and goal, in most cases these two semantics give the same results. In some cases, these results differ from those sanctioned by classical logic, but are similar to those observed in psychological studies of human reasoning. However, as we have seen at the beginning of the paper, the selection task presents a challenge, which we will discuss again in section 10.2.

6.4 Two kinds of proof procedures

The standard proof procedure for logic programming, used in Prolog for example, generates a *search tree*, whose *nodes* are labelled by goals. The *root* of the tree is labelled by the initial goal. Given any node in the tree, labelled by a goal, say:

$$B \,\&\, B_1 \,\&... \&\, B_n$$

backward reasoning proceeds by first selecting a sub-goal for solution, say *B* for simplicity.

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, but still “weak”, problem-solving strategies, such as choosing the most constrained sub-goal first (the one that has fewest candidate solutions), can also be used (Kowalski, 1974/79).

If the selected sub-goal *B* is a positive, atomic goal, then backward reasoning continues by looking for a clause whose conclusion matches *B*. In the propositional case, if *B* is identical to the conclusion of a clause *B if* B'_1 *&... &* B'_m then *backward reasoning* replaces the selected sub-goal by the conditions of the clause, obtaining a *child* of the node labelled by the new goal:

$$B'_1 \,\&... \&\, B'_m \,\&\, B_1 \,\&... \&\, B_n$$

In the more general case, the selected sub-goal needs to be unified with the conclusion of a clause, and the unifying substitution is applied to the new goal. A node has as many children as there are such clauses whose conclusion matches the selected sub-goal.

If the selected sub-goal *B* is the negation *not B’* of an atomic formula *B’* containing no variables, then *negation as failure* is applied, generating a subsidiary search tree with the same program, but with the initial goal *B’*. The selected sub-goal *not B’* succeeds if and only if the subsidiary search tree contains no solution, and the *only child* of the node is labelled by the new goal:

$$B_1 \,\&... \&\, B_n$$

A *solution* is a finite branch of the search tree that starts from the initial node and ends with a node labelled by the empty conjunction of sub-goals (where $n = 0$). Because the search tree may contain infinite branches, the proof procedure is semi-decidable.

Given a program and an initial goal, the search tree is generated and explored, to find a solution. Prolog uses a depth-first search strategy, exploring branches in the order in which clauses are written. Depth-first search is risky, because it can get lost down an infinite branch, when there are solutions lying around on alternative finite branches. However, when it works, it works very efficiently. But other search strategies, such as breadth-first, best-first and parallel search strategies, which are guaranteed to find a solution if one exists, are also possible.

If a search tree is too big, no search strategy can search it efficiently. Sometimes the problem is inherently difficult, and there is no alternative but to search the space as patiently as possible. But in many domains, as we have seen, the problem can be reformulated, using "stronger" knowledge, which results in a smaller, more manageable search tree.

The alternative to generating a search tree is to reason explicitly with the program completion, representing the current state of the search by a single formula, which corresponds to the expanding frontier of the search tree. Reasoning with the completion was introduced as a proof procedure for abductive logic programming by (Console et al, 1991) and extended by (Fung and Kowalski, 1997).

Backward reasoning is performed by replacing predicates with their definitions, resulting in a logically equivalent formula. For example:

Goal:	*you get help and you study late in the library*
Program:	*you get help if and only if*
	you press the alarm signal button or you shout loudly.
New goal:	*[you press the alarm signal button and you study late in the library] or*
	[you shout loudly and you study late in the library].

Reasoning with the completion represents the current state of the search to solve a goal as a disjunction of conjunctions of literals, say:

$$(A \,\&\, A_1 \,\&...\&\, A_\alpha) \text{ or } (B \,\&\, B_1 \,\&...\&\, B_\beta) \text{ or...or } (C \,\&\, C_1 \,\&...\&\, C_\nu).$$

Each disjunct corresponds to the goal at the end of an unextended branch in a search tree. The entire disjunction corresponds to the set of all goals at the current frontier of the search tree. The initial goal is represented by an initial disjunction with only one disjunct. Replacing an atom by its definition, called *unfolding*, corresponds to *replacing* a node at the tip of a branch by *all* of its children.

In the propositional case, unfolding a selected positive goal atom, say *B*, replaces it with its definition:

$$B \text{ if and only if } D_1 \text{ or...or } D_n$$

and distributes conjunction over disjunction, so that the new state of the search is represented by a new, logically equivalent formula of the form:

$$(A \,\&\, A_1 \,\&...\&\, A_\alpha) \text{ or } (D_1 \,\&\, B_1 \,\&...\&\, B_\beta) \text{ or...or } (D_n \,\&\, B_1 \,\&...\&\, B_\beta) \text{ or...or } (C \,\&\, C_1 \,\&...\&\, C_\nu).$$

In the special case where the definition has the form:

B if and only if true

then the new formula is simplified to:

$(A \,\&\, A_1 \,\&...\&\, A_\alpha)$ *or* $(B_1 \,\&...\,\&\, B_\beta)$ *or...or* $(C \,\&\, C_1 \,\&\, ... \,\&\, C_\nu)$.

In the special case where the definition has the form:

B if and only if false

then the new formula is simplified to:

$(A \,\&\, A_1 \,\&...\&\, A_\alpha)$ *or ...or* $(C \,\&\, C_1 \,\&\, ... \,\&\, C_\nu)$.

In the propositional case, the initial goal *succeeds* if and only if the formula *true* is derived by means of a finite number of such equivalence-preserving inference steps. The goal *fails finitely* if and only if the formula *false* is derived[5].

If the selected atom is a negative literal, say *not B'*, then a subsidiary derivation is initiated with the initial goal *B'* (viewed as a disjunction with only one disjunct, which is a conjunct with only one conjunct). If the new initial goal fails finitely, then the selected atom is replaced by *true* and the new formula is simplified to:

$(A \,\&\, A_1 \,\&...\&\, A_\alpha)$ *or* $(B_1 \,\&...\,\&\, B_\beta)$ *or...or* $(C \,\&\, C_1 \,\&\, ... \,\&\, C_\nu)$.

If the initial goal succeeds, then the selected atom is replaced by *false* and the new formula is simplified to:

$(A \,\&\, A_1 \,\&...\&\, A_\alpha)$ *or ...or* $(C \,\&\, C_1 \,\&\, ... \,\&\, C_\nu)$.

For example, consider the problem of determining whether:

The apple is red.

follows from the program:

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

In the completion semantics this can be represented in the form[6]:

[5] In the general case, the process of matching goals with conclusions of definitions introduces equalities, which represent the instantiations of variables need to solve those goals. The initial goal succeeds if and only if a consistent set of equalities is derived.

[6] This is not the real completion, which makes the stronger closed world assumption that nothing is illuminated by a red light. In fact, it would be more appropriate to represent the predicate *is illuminated by a red light* by an abducible predicate, to which the closed world assumption does not apply. The form of the completion used in this example has been chosen as a compromise, to simplify the example.

Goal$_1$:	*The apple is red.*
Program:	*An object is red if and only if*
	the object looks red and not the object is illuminated by a red light.
	The apple looks red if and only if true
	The apple is illuminated by a red light if and only if false

Using the completion, it is possible to derive the following sequence of formulae:

Goal$_2$:	*The apple looks red and not the apple is illuminated by a red light.*	
Goal$_3$:	*not the apple is illuminated by a red light.*	
	Subsidiary goal$_1$:	*the apple is illuminated by a red light.*
	Subsidiary goal$_2$:	*false*
Goal$_4$:	*true*	

The example shows that reasoning with the if-halves of clauses and generating a search tree simulates reasoning explicitly with the completion. Indeed, reasoning informally it can be hard to tell the two proof procedures apart.

7 The relationship between logic programs and production rules.

Viewing logic programs and production rules informally, it can also be hard to tell them apart, because in many cases they generate the same goal-reduction behaviour. Indeed, Simon (1999) includes Prolog, along with ACT-R, "among the production systems widely used in cognitive simulation". Stenning and van Lambalgen (2004, 2005, 2008) also observe that forward chaining with production rules of the form:

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

behaves like backward reasoning with clauses of the form:

G if C and H.

The production rule and the clause both behave as the same goal-reduction procedure:

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

The production rule can be understood as representing the procedure *subjectively* (or *intentionally*), in terms of the causal effect of the goal *G* on the agent's state of mind. The clause, on the other hand, 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 effects. 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). In Appendix B, I argue that the completion semantics helps to explain the switching and truncation of conditionals observed in many psychological studies, as discussed by Politzer and Bonnefon (2006)

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.

can be understood as a subjective representation of the goal-reduction procedure:

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

which can 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 implicit and unstated conditions also hold. This implicit condition can be stated explicitly using negation as failure:

you will go home for the weekend if
you have the bus fare and you catch a bus and
not something goes wrong with the bus journey.

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

This representation with an extra condition, using negation as failure, does not attempt to quantify or even to qualify the associated degree of uncertainty. However, as we will see later, in abductive logic programming (ALP) it can be quantified by using instead an abducible predicate, say *the bus 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 bus journey is successful.

the bus journey is successful (with probability .95).
something goes wrong with the bus journey (with probability .05).

In the meanwhile, it is worth noting that the ALP formulation provides a link with Bayesian networks (Pearl, 1988). David Poole (1993, 1997) has shown, in general, that assigning a probability *p* to a conditional *A if B* is equivalent to assigning the same probability *p* to an abducible predicate, say *normal*, in the clause *A if B and normal.* In particular, he has shown that abductive logic programs with such probabilistic abducible predicates have the expressive power of discreet Bayesian networks. A similar result, but with a focus on using statistical learning techniques to induce logic programs from examples, has also been obtained by Taisuke Satoh (1995). In recent years this has given rise to a booming research area combining probability, learning and logic programming (De Raedt and Kersting, 2003).

The relationship between logic programs that are used to reason backwards and production rules that are used to represent goal-reduction procedures does not solve the problem of the relationship between logic programs and production systems in general. In particular, it does not address the problem of understanding 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 goal-reduction procedure, to solve the goal of covering yourself with an umbrella by getting an umbrella when it rains.

I will argue that stimulus-response production rules are better understood as *maintenance goals*, which are treated as integrity constraints in abductive logic programming.

8 Integrity constraints

In conventional databases, integrity constraints are used passively to prevent prohibited updates. But in abductive logic programming, they are like integrity constraints in active databases, where they both prevent prohibited updates and perform corrective actions to ensure that integrity is maintained. Production rules, used as stimulus-response associations, can be understood as integrity constraints of this kind.

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.

This is identical in syntax to the condition-action rule, except 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, the integrity constraint has the syntax of a logical implication.

In fact, integrity constraints are just a particular species of goal; and, like other kinds of goals, they can be sentences of first-order logic. They also have the same semantics as goals in logic programs (Godfrey *et al*, 1998). This includes both the credulous and sceptical versions of the minimal model semantics, in which integrity constraints are sentences that are true in minimal models of a logic program. It also includes the completion semantics, in which they are sentences logically entailed by the if-and-only-if completion of the program.

However, it is also natural to understand the semantics of integrity constraints in consistency terms: An integrity constraint holds if and only if it is *consistent* with respect to the program. The consistency view comes in two flavours: In the minimal model semantics, the consistency view is equivalent to the credulous semantics: An integrity constraint is consistent with respect to the minimal model semantics of a logic program if and only if it is true in some minimal model of the program. In the completion semantics, an integrity constraint is satisfied if and only if it is consistent with the completion of the program relative to classical model theory. We will see later in section 10.2 that the consistency view of constraint satisfaction seems to be necessary in the selection task.

The difference between integrity constraints and conventional goals is partly terminological and partly pragmatic. In problem-solving, a conventional goal is typically an *achievement goal*, which is a one-off problem to be solved, including the problem of achieving some desired state of the world. In contrast, a typical integrity constraint is a *maintenance goal* that persists over all states of the world. In the context of intelligent agents, the term *goal* generally includes both *maintenance goals* and *achievement goals*.

Maintenance goals typically have the form of conditionals and can be hard to distinguish from clauses[7]. The distinction between clauses and maintenance goals can be better understood by viewing them in terms of the data base distinction 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, relationships can also be defined by more general 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:

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

The existential quantifier in the conclusion of the sentence means that the sentence cannot be used to define data, but can only be used to constrain it. 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, generating an appropriate value for Y.

Obviously, the capability to make such an update lies outside the powers of even an active database. However, it is a feature of many intelligent agent systems, such as Agent0 (Shoham, 1991). Whereas actions in production systems can be executed as soon as all of their conditions are satisfied, actions in Agent0 have associated times, and they can be executed when their associated times equal the actual time.

An intelligent agent might use integrity constraints to maintain the integrity of its “knowledge base”, in much the same way that an active database system uses integrity constraints to maintain

[7] In this respect, they resemble the conditionals in the selection task, which can be interpreted descriptively or deontically.

the integrity of its data. However, whereas backward reasoning is adequate for solving achievement goals, maintaining integrity is more naturally performed by combining backward and forward reasoning (Kowalski, Sadri and Soper, 1987):

> Integrity checking is triggered by an observation that updates the agent's knowledge base. Forward reasoning is used to match the observation either with a condition of an integrity constraint ***I*** or with a condition of a clause ***C***. It proceeds by using backward reasoning to verify any remaining conditions of the integrity constraint ***I*** or clause ***C***; and then, if these condtions are satisfied, it derives the conclusion. If the derived conclusion is the conclusion of an integrity constraint ***I***, then the conclusion is a new goal, typically an achievement goal. However, if it is the conclusion of a clause ***C***, then it is treated as a derived update and processed by further forward reasoning.

An achievement goal can be solved by backward reasoning, as in normal logic programming. But, instead of failing if a sub-goal cannot be reduced to further sub-goals, if the sub-goal is an action, then an attempt can be made to make it true by executing it successfully[8]. The result of the attempt is added as a new update to the knowledge base.

This informal description of forward and backward reasoning with clauses and forward reasoning with integrity constraints is compatible with different semantics and can be formalised both in terms of search trees and in terms of reasoning with the completion. In both cases, it gives a logical semantics to production rules of the stimulus-response variety and it facilitates combining them with logic programs. It also facilitates the modelling of agents that combine the ability to plan pro-actively with the ability to behave reactively.

9 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 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, whose desires (or goals) are represented by integrity constraints, and whose intentions are collections of actions to be performed in the future.

Arguably, the most influential of the BDI agent models is that of Rao and Georgeff (1991), 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 bore little resemblance to their logical specifications. AgentSpeak abandoned the attempt to relate the modal logic specifications with their procedural implementations, observing instead that "…one can view agent programs as multi-threaded interruptible logic programming clauses"

However, this view of AgentSpeak in logic programming terms is restricted to the procedural interpretation of clauses. In fact, procedures in AgentSpeak are much closer to production rules than they are to clauses in logic programming. Like production systems, AgentSpeak programs have both a declarative and a procedural component. The declarative component contains both

[8] If the abducible sub-goal is an observable predicate, then the agent can actively attempt to observe whether it is true or false. Active observations turn the simplified formulation of the selection task discussed at the beginning of the paper into one that more closely resembles standard formulations, in which the subject is asked to perform an action.

belief literals (atoms and negations of atoms) and *goal atoms*, whereas the procedural component consists of *plans*, which are an extension of production rules. Plans are embedded in a cycle similar to the production system cycle, and have 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. Like production rules, plans are executed in the direction in which they are written, by a kind of forward chaining. Production systems are the special case where the set of goals *G* is empty.

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.

Notice that the first two plans are goal-reduction rules, the third plan is a stimulus-response rule, and the fourth plan behaves like a logical implication used to reason forwards.

Like rules in production systems, plans in AgentSpeak do not have a declarative reading. 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 $\Leftarrow$ *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 replaced by predicates with explicit associated times. The corresponding clause is identical in behaviour to the plan, but also has a declarative reading. Using an (overly) simple, explicit representation of time[9], the clauses corresponding to the 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, plans in which the triggering event is the addition or deletion of a belief can be represented by means of an integrity constraint. For example, the integrity constraint corresponding to the third plan above is:

> *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. The fourth plan 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.

10 Abductive logic programming (ALP)

Abductive logic programming (Kakas et al, 1998) extends normal logic programming by combining closed predicates that are defined in the conclusions of clauses with abducible (open or undefined) predicates that occur in the conditions, but not in the conclusions of clauses. Abducible predicates are instead constrained, directly or indirectly, by integrity constraints.

10.1 A brief introduction to ALP

In general, given a logic program ***P*** and integrity constraints ***I***, the *abductive task* is given by a problem ***G***, which is either an achievement goal or an observation to be explained. The task is solved by finding a set Δ of atomic sentences in the abducible predicates, such that:

> Both ***I*** and ***G*** hold with respect to the extended, normal logic program $\boldsymbol{P} \cup \Delta$.

Δ is called an *abductive explanation* of ***G.***

This characterisation of abductive logic programming is compatible with different semantics[10] defining what it means for a goal or integrity constraint to hold with respect to a normal logic

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

program. It is compatible, in particular, with the minimal model semantics, the completion semantics and the consistency view of constraint satisfaction. It is also compatible with different proof procedures, including both search tree proof procedures and proof procedures that reason explicitly with the completion.

Consider the following abductive logic program (with a much simplified representation of time), in which the integrity constraint is a denial:

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 constraint:	*not (it rained and the sun was shining*)
or equivalently	*if it rained and the sun was shining then false.*
Observation:	*Grass is wet.*

In abductive logic programming, reasoning backwards from the observation (treated as a goal), it is possible to derive two alternative explanations of the observation:

it rained,
the sprinkler was on.

Depending on the proof procedure, the two explanations are derived either on different branches of a search tree or as a single formula with an explicit disjunction.

Suppose that we are now given the additional observation:

Observation: *the sun was shining.*

Backward reasoning with the new observation is not possible. But, depending on the proof procedure, as we will see in the next section, the new observation and the integrity constraint reject the hypothesis *it rained*, leaving *the sprinkler was on* as the only acceptable explanation of the observation that the *Grass is wet.*

10.2 Proof procedures for ALP

The standard proof procedure for ALP augments the search tree proof procedure for normal logic programming in two ways. First, it includes all integrity constraints in the initial goal, expressed in the form of denials. Second it accumulates abductive hypotheses as they are needed to solve the initial goal, and it adds these to the program as the search to find a solution continues.

[10] Sometimes the set Δ is allowed to contain the negations of atomic sentences in the abducible predicates, in which case there is an additional requirement that Δ be consistent.

For example, the initial goal, at the root of the search tree, needed to explain the observation that the *Grass is wet* in the previous section, would be expressed in the form[11]:

Grass is wet and not (it rained and the sun was shining)

Selecting the first sub-goal *Grass is wet*, backward reasoning generates two alternative branches in the search tree, with two children of the root node:

it rained and not (it rained and the sun was shining)
the sprinkler was on and not (it rained and the sun was shining)

On each branch, the proof procedure solves the abducible sub-goal by adding it to the set of hypotheses *Δ*:

not (it rained and the sun was shining)   *Δ = {it rained}*
not (it rained and the sun was shining)   *Δ = {the sprinkler was on}*

On both branches, negation as failure is applied to the remaining negative sub-goal, generating subsidiary search trees with new initial goals, and with the program augmented by the hypotheses generated so far:

it rained and the sun was shining   *Δ = {it rained}*
it rained and the sun was shining   *Δ = {the sprinkler was on}*

Given no additional information, both subsidiary search trees fail to contain a solution (in both cases *the sun was shining* fails). So the negative goals on the two branches of the original search tree succeed, resulting in two alternative solutions of the initial goal:

Δ = {it rained}   *Δ = {the sprinkler was on}*

However, given the additional observation *the sun was shining* added to the program, the first subsidiary search tree, with the goal *it rained and the sun was shining*, now contains a solution, because both sub-goals now succeed, and the corresponding negative goal now fails, leaving only the one solution to the initial goal:

Δ = {the sprinkler was on}

Although the search tree proof procedure can be extended to include forward reasoning with clauses and integrity constraints, it is easier to illustrate forward reasoning with completion proof procedures instead. There are different proof procedures that use the completion explicitly.
For example, the proof procedure of Console et al (1991) represents integrity constraints as denials and uses the consistency view of constraint satisfaction, whereas the IFF proof procedure of Fung and Kowalski (1997) and the SLP proof procedure of Kowalski, R., Toni, F. and Wetzel, G. (1998) represent integrity constraints as implications. The IFF and SLP proof procedures differ mainly in their implementation of constraint satisfaction. IFF implements the theorem-hood view, and SLP implements the consistency view.

[11] Strictly speaking, the denial should be formulated as a negative literal, by introducing an auxiliary predicate, but this is just a technical requirement, which can be avoided by slightly complicating the syntax of conditions of clauses, as in this example.

Given the problem of explaining the observation that the *Grass is wet,* the IFF and SLP proof procedures behave identically. The initial goal and program are expressed in the form:

Goal$_1$: *Grass is wet and (if it rained and the sun was shining then false)*
Program: *Grass is wet if and only if it rained or the sprinkler was on.*

Unfolding the first sub-goal, replacing it by its definition and distributing conjunction over disjunction, we obtain the equivalent new goal:

Goal$_2$: *[it rained and (if it rained and the sun was shining then false)] or*
[the sprinkler was on and (if it rained and the sun was shining then false)]

Forward reasoning within the first disjunct of the new goal adds the conclusion *if the sun was shining then false* to the disjunct:

Goal$_3$: *[it rained and (if it rained and the sun was shining then false) and*
(if the sun was shining then false)] or
[the sprinkler was on and (if it rained and the sun was shining then false)]

The proof procedure now terminates, because there are no further inferences that can be performed. It is a property of the IFF and SLP proof procedures that both of the hypotheses *it rained* and *the sprinkler was on* entail the observation, and the integrity constraint is both entailed by and consistent with the completion of the program extended with the completion of the hypotheses.

If the new observation *the sun was shining* is now added to both disjuncts of the current state of the goal, we obtain:

Goal$_4$: *[it rained and (if it rained and the sun was shining then false) and*
(if the sun was shining then false) and the sun was shining] or
[the sprinkler was on and (if it rained and the sun was shining then false) and
the sun was shining]

Forward reasoning with the new observation in both disjuncts, adds *false* to the first disjunct and *if it rained then false* to the second disjunct. Simplifying the first disjunct to *false* leaves only the second disjunct in the form:

Goal$_5$: *[the sprinkler was on and (if it rained and the sun was shining then false) and*
the sun was shining and (if it rained then false)]

The proof procedure terminates again. This time, the hypothesis that *the sprinkler was on* is now the only hypothesis that entails the observation and satisfies the integrity constraint. In this example, the same proof procedure implements both the theorem-hood and the consistency views of constraint satisfaction.

The IFF and SLP proof procedures differ, however, on the deontic version of the selection task. Because IFF implements the theoremhood view of constraint satisfaction and uses integrity constraints only for forward reasoning, it cannot derive *if P then false* (i.e. *not P) from if Q then false* (i.e. *not Q) and the integrity constraint if P then Q.* But SLP can perform this inference, because it implements a general-purpose resolution method to check consistency.

The IFF proof procedure is sound and, with certain modest restrictions on the form of clauses and integrity constraints, complete with respect to the completion of the program in the Kunen (1987) three-valued semantics. SLP is sound but not complete, because consistency is not semi-decidable. However, this is not an argument against the consistency view - any more than the fact that there is no proof procedure for proving all truths of arithmetic is an argument against the notion of truth in arithmetic. (See, in particular, the Horn clause program for arithmetic in Appendix A.)

11 ALP agents

The notion of ALP agent, in which abductive logic programming is embedded as the thinking component of an observation-thought-decision-action cycle, was introduced in (Kowalski, 1995) and was developed further in (Kowalski and Sadri, 1999; Kowalski 2001, 2006). It is also the basis of the KGP (Knowledge, Goals and Plans) agent model of (Kakas el al, 2004). The ALP proof procedure of (Kowalski and Sadri, 1999) is the IFF proof procedure, whereas the informal proof procedure described in this paper and in (Kowalski 2001, 2006) extends IFF by incorporating forward reasoning with clauses, as in the integrity checking method of (Kowalski, Sadri, & Soper, 1987). The proof procedure of KGP extends IFF with constraint handling procedures.

The observation-thought-decision-action cycle of an ALP agent is similar to the production system and AgentSpeak cycles. The agent's beliefs are represented by clauses. Achievement and maintenance goals, as well as prohibitions, are represented by integrity constraints. Observations (or hypotheses that explain observations) and actions are represented 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 the execution of actions that need to be performed only later in 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 you 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. Moreover, the second and third sentences have the same conclusion *you get help.* However, despite their similar syntax, the first two sentences would function as *maintenance goals* and would be represented by integrity constraints in the ALP agent model. The last 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.

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

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 backward reasoning 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. 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 third 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 action is urgent, it interrupts the process of studying late in the library, which can 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* $\Leftarrow$ *!you study late in the library*
> \+*there is an emergency: true* $\Leftarrow$ *!get help.*
> \+*!get help: true* $\Leftarrow$ *press the alarm signal button.*
> \+ *?there is a fire: there is an emergency* $\Leftarrow$ *true.*
> \+ *there is a fire: true* $\Leftarrow$ +*there is an emergency.*
> \+*! there is an emergency: true* $\Leftarrow$ *! there is a fire.*

Because AgentSpeak plans do not have a uniform logical reading, they can not easily be used to represent weak domain-general knowledge, as is possible in logic programming. On the one hand, this is a limitation, but on the other hand it probably encourages programmers to write more efficient domain-specific problem-solving methods.

12 Decision-making in ALP agents – another role for probability

The ALP agent observation-thought-decision-action cycle includes a decision-making component to choose between alternatives courses of action, including alternative ways of solving the same goal. Different alternatives may solve an initial goal with different degrees of utility, as well as have other desirable or undesirable consequences as side effects.

This decision-making process is further complicated by the uncertainty of the consequences of an agent's candidate actions, due to the uncertain behaviour of other agents and the uncertainty of the environment. Thus the problem of deciding between different courses of actions requires consideration of both utility and uncertainty. According to the norms of classical decision theory, a rational agent should choose a course of actions that optimises overall expected utility, including the utility of any side effects, and taking into account any associated uncertainties.

Decision theory is a normative ideal, which requires a large body of knowledge and huge computational resources. As a consequence, it is often more practical for an agent to employ heuristic strategies that approximate the normative ideal. One of the simplest of these strategies is to use a fixed priority ordering of goals and beliefs, to give preference to one set of actions over another.

Consider, for example, the following two 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 the car home.

Both alternatives can be used to solve the goal of going home, but 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 the bus journey is successful.
you will go home for the weekend if you have a car and you drive the car home and the car journey is successful.

the bus journey is successful. (with probability .95)
the car journey is successful. (with probability .80)

To do a full-scale decision-theoretic analysis, in addition to determining the probabilities, it would also 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[12]. 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 calculating the probabilities and utilities and combining them to find the optimal solution. But, the result might be equivalent to just giving preference to the use of one alternative over the other. This could be done, for example, just using normal logic programming clauses, Prolog-fashion in a fixed order. For example, in the order:

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 journey.

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

13 Conclusions

[12] In classical Decision Theory, the possible outcomes of candidate actions are given in advance. The extension of the IFF proof procedure to include forward reasoning with clauses makes it possible to derive consequences of candidate actions to help in deciding between them (Kowalski, 2006), using clauses of the form *effect if cause.* The same clauses can also be used backwards to generate plans to achieve desired *effects* by reducing them to possible *causes.*

I have argued that the two main kinds of conditionals used in AI, namely production rules and logic programming clauses, can be combined naturally in the ALP agent model and can simulate human performance on some reasoning tasks. Because production systems have been used widely as a cognitive architecture, but have not been much used to model human reasoning, there are grounds for believing that the ALP agent model may have value as a general cognitive architecture.

Psychological studies of human reasoning are a test and a challenge for agent models developed in Artificial Intelligence. Although the ALP agent model performs well on some of these tests, the basic model needs to be extended, to include forward reasoning with clauses and to implement the consistency semantics of integrity constraints. I believe that these extensions, and possibly others that might be needed to simulate human reasoning in other psychological experiments, will also be useful for computer applications in AI.

Acknowledgements

Many thanks to Fariba Sadri, Michiel van Lambalgen and Keith Stenning for helpful discussions, and to Keith Clark, Luis Pereira, Keith Stenning and Guy Politzer for useful comments on an earlier draft of this paper. Thanks too to Mike Oaksford for the invitation to contribute to this volume.

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.

Bonnefon, J.-F. and Politzer, G. (2007) Pragmatic Conditionals, Conditional Pragmatics, and the Pragmatic Component of Conditional Reasoning. *This volume.*

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.

Console, L., Theseider Dupre, D. and Torasso, P. (1991) On the relationship between abduction and deduction. *Journal of Logic and Computation. 1(5)* 661-690

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

De Raedt, L. and Kersting, K. (2003) Probabilistic Logic Learning. In: SIGKDD *Explorations 5/1* 31-48.

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.

van Emden, M. (2006) The Early Days of Logic Programming: A Personal Perspective
The Association of Logic Programming Newsletter, Vol. 19 n. 3, August 2006.
http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/newsletter/aug06/

van Emden, M. and Kowalski, R. (1976) The Semantics of Predicate Logic as a Programming Language *JACM* , Vol. 23, No. 4, 733-742.

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.

Godfrey, P., Grant, J., Gryz, J. and Minker, J. (1998) Integrity constraints: semantics and applications. In *Logics for databases and information systems.* Kluwer, Norwell, MA, USA. 265 – 306.

Kakas, A., 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.

Kakas, A., Mancarella, P., Sadri, S., Stathis, K. and Toni, F. (2004) *The KGP model of agency*, ECAI04, General European Conference on Artificial Intelligence, Valencia, Spain, 33-37.

Kowalski, R. and Kuehner, D. (1971) Linear Resolution with Selection Function. *Artificial Intelligence*, Vol. 2, 227-60. Reprinted in *Anthology of Automated Theorem-Proving Papers*, Vol. 2, Springer-Verlag, 1983, 542-577.

Kowalski, R., (1973) Predicate Logic as Programming Language. Memo 70, Department of Artificial Intelligence, Edinburgh University. Also in *Proceedings IFIP Congress*, Stockholm, North Holland Publishing Co., 1974, 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. (1974) *Logic for Problem Solving.* DCL Memo 75, Department of Artificial Intelligence, U. of Edinburgh. Expanded edition published by North Holland Elsevier 1979. Also at http://www.doc.ic.ac.uk/~rak/.

Kowalski, R. (1995) Using metalogic to reconcile reactive with rational agents.
In: Meta-Logics and Logic Programming (K. Apt and F. Turini, eds.), MIT Press

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. and Sergot, M. (1986) A Logic-based Calculus of Events. In New Generation Computing, Vol. 4, No.1, 67-95. Also in The Language of Time: A Reader (eds. Inderjeet Mani, J. Pustejovsky, and R. Gaizauskas) Oxford University Press. 2005.

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.

Kowalski, R., Toni, F. and Wetzel, G. (1998) Executing Suspended Logic Programs. *Fundamenta Informatica 34 (3),* 1-22.

Kunen, K. (1987) Negation in logic programming. *Journal of Logic Programming 4:4* 289 - 308

Laird, J.E., Newell, A. and Rosenblum, P. S. (1987) SOAR: an architecture for general intelligence. *Artificial Intelligence,* 33:1, 1 – 64.

Lloyd, J W, Topor, R W (1984) Making PROLOG more expressive. *Journal of Logic Programming.* Vol. 1, no. 3, 225-240.

McCarthy, J. (1958) Programs with common sense, *Symposium on Mechanization of Thought Processes*. National Physical Laboratory, Teddington, England.

McCarthy, J. (1980) Circumscription - A form of non-monotonic reasoning. *Artificial Intelligence*, 13:27-39.

McDermott, D. and Doyle, (1980) Nonmonotonic logic I," *Artificial Intelligence*, 13:41-72.

Minsky, M. (1974) A Framework for Representing Knowledge. Technical Report: AIM-306, Massachusetts Institute of Technology, Cambridge, MA.

Moore, R. C. (1985). Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75-94.

Newell, A., Shaw, J. C., and Simon, H.A. (1957) Empirical explorations of the logic theory machine. Proc. Western Joint Comp. Conf, 218-239.

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

Newell, A. (1990). *Unified theories of cognition.* Harvard University Press Cambridge, MA, USA.

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., & Georgeff, M. P. (1991). *Modeling rational agents with a BDI-architecture*. Second International Conference on Principles of Knowledge Representation and Reasoning. 473-484.

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.

Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13:81-132.

Robinson, J. A. (1965) A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM*, 12(1) 23 – 41.

Russell, S.J. & Norvig, P. (2003) Artificial Intelligence: A Modern Approach (2nd ed.), Upper Saddle River, NJ: Prentice Hall.

Sagonas, K., Swift, T. & Warren, D. (1994) XSB as an efficient deductive database engine. In ACM SIGMOD, ACM SIGMOD Record archive
Volume 23 , Issue 2 442 - 453

Sato, T. (1995) A statistical learning method for logic programs with distribution semantics
In *Proceedings of the 12th International Conference on Logic Programming.* MIT Press. 715-729.

Shoham, Y. (1991) *AGENT0:* A Simple Agent Language and Its Interpreter. In *Proceedings of the Ninth National Conference on Artificial Intelligence* (AAAI-91), AAAI Press/MIT Press, Anaheim, California, USA, 704-709.

Shortliffe, E. (1976) *Computer-based Medical Consultations: MYCIN.* North Holland Elsevier.

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

Sperber, D., Cara, F., & Girotto, V. (1995). Relevance theory explains the selection task. *Cognition, 52,* 3-39.

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), 919–96

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

Thagard, P. (2005) *Mind: Introduction to Cognitive Science.* Second Edition. MIT Press.

Wason, P. C. (1964). Reasoning. In Foss, B. (Ed.), New Horizons in Psychology. Harmondsworth: Penguin Books

Wason, P. C. (1968) 'Reasoning about a rule', The Quarterly Journal of Experimental Psychology, 20:3, 273 - 281

Winograd, T. (1971) *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language* MIT AI Technical Report 235, February 1971. Also published as a full issue of the journal *Cognitive Psychology* Vol. 3 No 1, 1972, and as a book, *Understanding Natural Language* (Academic Press, 1972).

Appendix A The minimal model semantics of logic programming

A.1 Horn clause programs

The minimal model semantics is consistent with the declarative reading of clauses as implications in classical logic. However, in programming and database contexts and with Horn clause programs ***P***, the semantics is also much simpler than in classical logic. In such cases, because clauses always have positive conclusions, it is natural to regard the set of clauses ***P*** as defining the set of positive, atomic sentences ***A*** that are logically implied by ***P***.

This relationship between ***P*** and an atomic sentence ***A*** can be understood purely in terms of classical, logical consequence:

> ***A*** is true in every model of ***P***.

However, sets of Horn clauses ***P*** have the special property (van Emden and Kowalski, 1976) that there exists a *unique minimal model* ***M*** of ***P*** such that:

> For every atomic sentence ***A,***
> ***A*** is true in every model of ***P*** if and only if ***A*** is true in ***M***.

This minimal model ***M*** can be represented syntactically as the set of all atomic sentences ***A*** that are true in ***M***. The model ***M*** is *minimal* in the sense that it is the smallest set of atomic sentences that is a model of ***P***.

For example, the Horn clauses, which define equality, addition and multiplication for the natural numbers:

$$X + 0 = X$$
$$X + (Y + 1) = (Z + 1) \textit{ if } X + Y = Z$$
$$X \cdot 1 = X$$
$$X \cdot (Y + 1) = W \textit{ if } X \cdot Y = Z \textit{ and } Z + X = W$$

have a unique minimal model, which is the standard model of arithmetic. In contrast, the completion of the Horn clause program is equivalent to the Peano axioms of arithmetic, without the axioms of induction.

The minimal model can be constructed in the limit, simply by repeatedly applying *modus ponens* and universal instantiation in all possible ways to all the clauses in ***P.*** This model construction property can also be interpreted as the completeness property of forward reasoning for atomic consequences. Backward reasoning, in the form of SLD-resolution (Kowalski, 1973/1974), which is equivalent to the search tree proof procedure, is also sound and complete for atomic consequences of Horn clauses.

A.2 The minimal model semantics of negation

The simplest case is that of a Horn clause program ***P*** and the negation *not G* of a positive atomic sentence *G*. According to the minimal model semantics:

> *not G holds* if and only if *not G* is *true* in the minimal model ***M*** of ***P***.

More generally, for any sentence *G*, including any sentence of first-order logic:

G holds if and only if *G* is *true* in ***M*.**

Backward reasoning and negation as infinite failure are sound and complete inference rules for this semantics. In other words:

G succeeds if and only if *G* is *true* in ***M*.**

This property depends upon the rewriting of first-order formulae into goals that are conjunctions of literals augmented with auxiliary clauses, as mentioned in section 6.1.

The minimal model semantics can be extended in a number of different ways to the more general case of normal logic programs ***P***, which contain negative literals in the conditions of clauses. One of the simplest and most natural of these extensions interprets *not G* literally as *G does not* hold and expands ***P*** with a set *Δ* of true sentences of the form *not G*. The phrase *does not hold* can be treated as a positive auto-epistemic predicate, so that the expansion ***P***∪*Δ* can be treated as a Horn clause program, which has a unique minimal model ***M***:

M is a *model* of ***P*** if all the sentences in *Δ* are true in ***M***.[13]

It is common to restrict auto-epistemic expansions *Δ* to ones that are total or maximal.[14] For simplicity, in this paper, I refer to this semantics and its variants simply and collectively as the *minimal model semantics* of negation as failure.

A program with negation as failure can have several such minimal models. According to the *credulous* version of this semantics,

a goal holds if and only if it is true in some minimal model.

According to the *sceptical* version of the same semantics,

a goal holds if and only if it is true in all minimal models.

In both cases, the goal can be any sentence of first-order logic, and backward reasoning and negation as infinite failure can be used to determine whether or not a goal holds.

[13] In other words, there is no *not G* in *Δ* such that *G* belongs to ***M***.

[14] If *Δ* consists of *all* such true sentences, then the resulting model is called a *stable model* (Gelfond and Lifschitz, 1990). A program can have many or no stable models. For example, the program {*p if not q, q if not p*} has two stable models {*p, not q*} and {*q, not p*}, but the program {*p if not p*} has no stable models.

Appendix B Conditionals in natural language

Part of the problem with understanding natural language conditionals is the problem of translating them into logical form. We saw an example of this in the motivating example at the beginning of the paper. However, the completion semantics suggests that there can be an added difficulty with choosing between different logical forms, corresponding to the two directions of the completion of a definition.

B.1 Switching the direction of conditionals

Consider, for example, the two clauses:

A if B.
A if C.

whose completion is

A if and only if B or C.

The only-if-half of the completion is implicit in the original set of two clauses. But it is also possible to write the two clauses in the *switched form,* in which the only-if half is explicit and the if-half is implicit:

B or C if A.

The disjunction in the conclusion of the switched form can be eliminated by using negative conditions, yielding the switched clauses:

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

For example, the two clauses for getting help in section 6.3 can be transformed into the following switched forms:

you have pressed the alarm signal button or you shouted loudly if you get help.
you have pressed the alarm signal button if you get help and you have not shouted loudly.
you shouted loudly if you get help and you have not pressed the alarm signal button.

The completion semantics, therefore, justifies four ways of representing the same conditional relationship: the if-form, the if-and-only-if form, the only-if form and the only-if form with negative conditions. Both the first and the fourth representation can be represented as logic programming clauses, and it can be difficult for a programmer to know which representation to use.

In applications such as fault diagnosis in AI, different kinds of representation are common. The first representation is common because it models causality in the form *effect if cause.* But this representation requires the use of abduction to explain observations. The fourth representation, which models causality in the switched form *cause if effect and not other-causes,* is also common because it requires only the use of deduction.

B.2 Truncated conditionals

Bonnefon and Politzer (2007) observe that many conditionals in natural language have a truncated form $A \rightarrow \Phi$ that, to be understood fully, needs to be put into a wider context of the form:

$$(A \ \& \ A_1 \ \&...\& \ A_\alpha) \ or \ (B \ \& \ B_1 \ \&... \ \& \ B_\beta) \ or...or \ (C \ \& \ C_1 \ \& \ ... \ \& \ C_\nu) \rightarrow \Phi$$

In this wider context, there may be both additional, unstated conditions $(A_1 \ \&... \ \& \ A_\alpha)$ and additional, unstated, alternative ways $(B \ \& \ B_1 \ \&...\& \ B_\beta) \ or...or \ (C \ \& \ C_1 \ \& \ ... \ \& \ C_\nu)$ of establishing the conclusion Φ.

But, as we have seen, this wider context is exactly a logic program, which can also be expressed in the completion form:

$$(A \ \& \ A_1 \ \&...\& \ A_\alpha) \ or \ (B \ \& \ B_1 \ \&... \ \& \ B_\beta) \ or...or \ (C \ \& \ C_1 \ \& \ ... \ \& \ C_\nu)) \leftrightarrow \Phi$$

The completion suggests the alternative possibility that a conditional might, instead, be a truncated form of the switched, only-if half $\Phi \rightarrow A$ of the wider context. I will come back to this possibility in B.3.

But, first, consider the following example of a simple truncated conditional, in logic programming form:

X flies if X is a bird.

The wider context is a more precise statement that includes extra conditions and alternative ways of flying. In most AI knowledge representation languages, it is common to express the conditional more precisely (although not as completely as in Bonnefon and Politzer's wider context) by lumping all of the extra conditions into a single condition, which is assumed to hold by default. For example:

X flies if X is a bird and not X is unable to fly.

Here the predicate *X is unable to fly* can be defined by other clauses, such as:

X is unable to fly if X is crippled.
X is unable to fly if X has not left its nest. etc.

which is equivalent to specifying other conditions for the clause. The minimal model and completion semantics both incorporate the closed world assumption that, unless there is evidence that a bird is unable to fly, then the bird is assumed to fly by default.

In general, the full alternative:

$$(A \ \& \ A_1 \ \&...\& \ A_\alpha) \rightarrow \Phi$$

which spells out all the missing conditions of a truncated conditional $A \rightarrow \Phi$ can be expressed in logic programming style with a default condition, which can be defined separately. For example in the form:

Φ if A & not exceptional
exceptional if not A_1 ….
exceptional if not A_α

Typically, the condition A of a truncated conditional $A \rightarrow \Phi$ is the most significant of the conditions A & A_1 &...& A_α , and the truncated conditional can be regarded as an *approximation* of the full alternative $(A$ & A_1 &...& $A_\alpha) \rightarrow \Phi$.

The precise form, say *A & not exceptional $\rightarrow \Phi$,* of a truncated conditional can be viewed as a solution of the *qualification problem*: How to suitably qualify a conclusion to adequately account for all its conditions, without specifying them all in advance and in detail. This precise form of the truncated conditional has the advantage that the conclusion can be qualified precisely from the outset by means of a single default condition, and exceptions to that condition can be spelled out separately in successively more precise formulations. The qualification problem is one aspect of the notorious *frame problem* in Artificial Intelligence, which has a similar solution.[15]

B.3 Truncated and switched conditionals

In B.1, I argued that the completion semantics explains why it may be difficult to determine the right direction for a conditional. This difficulty is compounded when a conditional is a truncated conditional $A \rightarrow \Phi$, which is part of a wider context

$(A$ & A_1 &...& $A_\alpha)$ or $(B$ & B_1 &... & $B_\beta)$ or...or $(C$ & C_1 & ... & $C_\nu) \rightarrow \Phi$

because the converse of the truncated conditional $\Phi \rightarrow A$ is itself a truncation of the switched form:

$\Phi \rightarrow (A$ & A_1 &...& $A_\alpha)$ or $(B$ & B_1 &... & $B_\beta)$ or...or $(C$ & C_1 & ... & $C_\nu)$

Understanding these two different ways in which a natural language conditional relates to its wider context might help to explain how people reason with conditionals. Consider the following example from (Politzer and Bonnefon, 2006).

Given the two sentences:

If an object looks red, then it is red.
This object looks red.

it is natural to draw the conclusion:

This object is red.

However, given the additional sentence:

[15] For example, in the event calculus (Kowalski and Sergot, 1986), the fact that a property is assumed to hold from the time it is initiated until it has been terminated is expressed in the form:
P holds at time T_2 if an event E initiates P at time T_1 and $T_1 < T_2$
and not (an event E' terminates P at time T and $T_1 < T < T_2$)

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

it is natural to withdraw the conclusion. But it is not natural to combine the two conditionals and draw the obviously false conclusion:

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

This way of reasoning can be explained if the two conditionals both refer to the same wider context, but the first one is switched. In logic programming, this wider context can be expressed in the clausal form:

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

In this analysis, the first inference that *this object is red* is explained by interpreting the first clause as the *only* clause that can be used in establishing that *an object looks red* and interpreting the first conditional as the switched, only-if half of the completion of the clause.

The withdrawal of the inference is explained by interpreting the new conditional as indicating that there is an additional alternative way of establishing that *an object looks red*, as represented by the second clause of the logic program. The only-if half of the two clause program no longer implies the withdrawn inference.

The failure to draw the obviously false conclusion is explained by the fact that in the wider context, the meaning of the two conditionals is actually the two clauses, which express alternative ways of establishing the same conclusion, and which do not combine to imply the false conclusion.

In summary, the example illustrates that a pair of natural language conditionals whose surface structure has the form:

$$\Phi \rightarrow A$$
$$B \rightarrow \Phi$$

Might actually have a very different deep structure, say:

$$(A \,\&\, \textit{not abnormal}) \text{ or } (B \,\&\, \textit{not abnormal'}) \leftrightarrow \Phi$$

or in clausal form:

$$\Phi \textit{ if } A \,\&\, \textit{not abnormal}$$
$$\Phi \textit{ if } B \,\&\, \textit{not abnormal'}$$