CHAPTER 12

Formalisation of Provability

The meta-language interpretation of "only-if" and its combination with the object language can be achieved by formalising the meta-language and amalgamating it with the object language. Such a combination of object language and meta-language produces a system of logic which is closer to natural language than the conventional systems which keep the two languages distinct. In natural language, however, the combination of object language and meta-language leads to such paradoxes as the self-referential sentence:

This sentence is false.

We shall see that the attempt to reconstruct the paradoxes in the amalgamated formal language leads instead to a true but unprovable sentence:

This sentence is unprovable.

The construction and proof of unprovability are based on those in Gödel's proof of the incompleteness of formal arithmetic [Gödel 1931]. Instead of the incompleteness of arithmetic, however, we have the impossibility of any attempt to completely formalise the notion of provability. The proof of incompleteness, moreover, is simpler for provability than it is for arithmetic.

Our purpose in combining the object language and meta-language, however, is primarily a practical one. The amalgamated language is more expressive and has greater problem-solving power than the object language alone. It provides essential facilities for such applications of logic programming as natural language understanding, database management, job control and editing of programs.

The amalgamated language combines object language and meta-language while preserving the normal semantics of logic. Thus all of the theory of problem-solving, formulated in the previous chapters for the object language alone, applies without change to the more powerful combination of object language and meta-language.

The combination of object language and meta-language is a special case of a more general construction. Given any two languages (i.e. systems of logic with their associated proof procedures) it may be possible to simulate the proof procedure of one language $L_1$ within the other $L_2$. The simulation is accomplished by defining in $L_2$ the binary relationship which holds when a conclusion can be derived from assumptions in $L_1$. Sentences in $L_1$ need to be named by terms in $L_2$ and the provability relation needs to be named by a binary predicate symbol, say
"Demonstrate", and defined by means of sentences Pr in L₂. Provided the
definition Pr correctly represents the provability relation of L₁,
simulation by means of Pr in L₂ is equivalent to direct execution of
the proof procedure of L₁. L₂, the language in which Pr simulates L₁,
is a meta-language for the object language L₁. To serve as meta-language,
L₂ needs to possess sufficient expressive power. For any object language,
the language of Horn clauses is already adequate.

There are a number of cases of special interest. In the case in which
the meta-language is restricted to the Horn clause subset of logic, but
the object language encompasses the whole standard form, the meta-
language improves its own problem-solving abilities by simulating the
more powerful object language. In general, a simple unsophisticated
problem-solver can improve itself by using simulation to behave like a
more sophisticated one.

In the case in which the object language and meta-language are
identical the single language augmented by the definition Pr of its own
provability relation is an amalgamation of an object language with its
meta-language.

**Correct representability**

The condition of correct representability is the same in principle for
the definition of the provability relation as it is for the definition of
the addition of natural numbers.

In order to define addition in logic, it is necessary to name numbers
by means of terms. The easiest way to name the non-negative integers, for
example, is by means of a constant symbol \( \emptyset \) for zero and a one-place
function symbol \( s \) for the successor function.

If \( t \) names the integer \( n \)
then \( s(t) \) names the integer \( n+1 \).

The following Horn clause definition correctly represents the addition
relation, named by the predicate symbol "Plus".

\[
\begin{align*}
\text{Plus1} & \quad \text{Plus}(\emptyset, x, x) \leftarrow \\
\text{Plus2} & \quad \text{Plus}(s(x), y, s(z)) \leftarrow \text{Plus}(x, y, z)
\end{align*}
\]

**Plus1-2 correctly represents** the addition relation in the sense that

whenever \( l, m \) and \( n \) are non-negative integers named by \( r \),
\( s \) and \( t \) respectively, the relationship \( l+m = n \) holds if-
and-only-if **Plus1-2** implies \( \text{Plus}(r, s, t) \leftarrow \).

Notice that correct representability does not require that

**Plus1-2** implies \( \neg \text{Plus}(r, s, t) \leftarrow \) when \( l+m = n \) does not
hold.

In order to define provability it is necessary to name sentences and
other expressions by means of terms. This can be accomplished in a
variety of ways and we shall not concern ourselves with the details here.
Correct representability

Given a representation of sentences by means of terms, a definition $Pr$ in a language $L_2$ correctly represents the provability relation, named "Demonstrate", of a language $L_1$ if-and-only-if

whenever $X$ and $Y$ are sentences of $L_1$ named by terms $X'$ and $Y'$ of $L_2$ respectively, conclusion $Y$ can be derived from assumptions $X$ in $L_1$ if-and-only-if conclusion $\text{Demonstrate}(X',Y')$ can be derived from assumptions $Pr$ in $L_2$.

Correct representability, however, does not require that $Pr$ implies $\neg\text{Demonstrate}(X',Y')$ in $L_2$ when $X$ does not imply $Y$ in $L_1$.

Given a language $L_1$, the construction of a definition which correctly represents its proof procedure is not a particularly difficult matter. Since proof procedures can be implemented by means of computer programs, they can be implemented by means of Horn clause programs in particular. Moreover, any Horn clause program which correctly implements a proof procedure correctly represents its provability relation.

A simple definition of a provability relation

We shall present the top-level of a Horn clause definition of the provability relation for a Horn clause language in which assumptions are regarded as programs and conclusions as collections of goals. In order to increase readability, we use lower case character strings, such as

\[
\text{prog, goals, sub,}
\]
as variables and ones beginning with an upper case character, such as

\[
\text{NIL, Zeus, A,}
\]
as constants.

The first clause of the program states that

any program demonstrates the solvability of an empty collection of goals.

The second clause, interpreted top-down, says that

- to demonstrate the solvability of a collection of goals:
  - select a goal;
  - find an appropriate procedure in the program;
  - rename the variables in the procedure so that they are distinct from the variables in the collection of goals;
  - match the selected goal with the head of the procedure;
  - add the body of the procedure to the rest of the goals;
  - apply the matching substitution to obtain a new collection of goals; and
  - demonstrate that the program solves the new collection of goals.
To complete the definition it is necessary to define the lower-level relations and to settle upon data structures for naming programs, goals, collections of goals and substitutions. Rather than define these in general, we shall present only an interface for the top-level with a simple data structure for the problem of the fallible Greek.

We shall name an atomic formula whose predicate symbol is named \( P \) and list of arguments is named \( t \) by the term

\[
\text{atom}(P,t).
\]

Bodies of procedures and collections of goals are named by lists of the names of the atomic formulae they contain. Programs and procedures are named by constants. The following clauses define the interface between the top-level of the definition of Demonstrate and the data structures for the problem of the fallible Greek.

\[
\begin{align*}
\text{Member}(F1, F) & \leftarrow \\
\text{Member}(F2, F) & \leftarrow \\
\text{Member}(F3, F) & \leftarrow \\
\text{Member}(F4, F) & \leftarrow \\
\text{Parts}(F1, \text{atom}(\text{Fallible},X,\text{NIL}), \text{atom}(\text{Human},X,\text{NIL})).\text{NIL}) & \leftarrow \\
\text{Parts}(F2, \text{atom}(\text{Human},\text{Turing},\text{NIL}), \text{NIL}) & \leftarrow \\
\text{Parts}(F3, \text{atom}(\text{Human},\text{Socrates},\text{NIL}), \text{NIL}) & \leftarrow \\
\text{Parts}(F4, \text{atom}(\text{Greek},\text{Socrates},\text{NIL}), \text{NIL}) & \leftarrow \\
\end{align*}
\]

The top-level goal is described by the clause

\[
\text{<- Demonstrate}(F, \text{atom}(\text{Fallible},X,\text{NIL}).\text{atom}(\text{Greek},X,\text{NIL})).\text{NIL}).
\]

The constant symbol \( X \) names the variable \( x \).

**Direct execution versus simulation**

Let \( Pr \) consist of the clauses D1-2 together with whatever lower-level clauses are needed to complete the definition of Demonstrate. Suppose that \( Pr \) correctly represents the provability relation of a language \( L_1 \) and is expressed in a language \( L_2 \) (which may be identical to \( L_1 \)). Correct representability guarantees that direct execution in \( L_1 \) and simulation in \( L_2 \) are equivalent and interchangeable.
Given sentences $X$ and $Y$ of $L_1$ named by terms $X'$ and $Y'$ respectively of $L_2$, direct execution of the proof procedure of $L_1$ to determine whether $Y$ can be derived from $X$ in $L_1$ is equivalent to simulation of $L_1$ by showing that $\text{Demonstrate}(X',Y')$ can be derived from $\text{Pr}$ in $L_2$.

The equivalence of direct execution and simulation is identical to the reflection principles investigated by Weyhrauch [1978].

Correct representability of the provability relation means that the object language and meta-language can cooperate to solve problems. A problem in the object language can be solved by simulation in the meta-language. Conversely, a problem of the form

$$\text{Demonstrate}(X',Y')$$

in the meta-language can be solved by showing that $Y$ can be derived from $X$

in the object language. This has the advantage that direct execution is generally more efficient than simulation in the meta-language.

Simulation in the meta-language, however, can be more powerful than direct execution. It may be possible, in particular, to replace several proofs of different, but similar, theorems in the object language by a single proof in the meta-language. As a trivial example, all of the problems below need to be solved separately in the object language, but can be solved once and for all in the meta-language.

$$\text{Mortal(Socrates)} \leftarrow \text{can be derived from}$$
$$\text{Human(Socrates)} \leftarrow \text{and}$$
$$\text{Mortal}(x) \leftarrow \text{Human}(x)$$

$$\text{Poisonous(\text{Boletus})} \leftarrow \text{can be derived from}$$
$$\text{Boletus(\text{Poisonous})} \leftarrow \text{and}$$
$$\text{Poisonous}(x) \leftarrow \text{Boletus}(x)$$

$$\text{Animal(\text{Puff})} \leftarrow \text{can be derived from}$$
$$\text{Dragon(\text{Puff})} \leftarrow \text{and}$$
$$\text{Animal}(x) \leftarrow \text{Dragon}(x)$$

In the meta-language it is possible with a single proof to show that

for any variable $x$, predicate symbols $P$ and $Q$, and term $t$ of the object language,

$$Q(t) \leftarrow \text{can be derived from}$$
$$P(t) \leftarrow \text{and}$$
$$Q(x) \leftarrow P(x).$$

The meta-language is more powerful than the object language in another sense. The object-level proof procedure can only show that

$$X$$

when both $X$ and $Y$ are given as input. The meta-level proof procedure,
however, can solve Demonstrate goals of any pattern of input and output.

Given, for example, an appropriate definition of what constitutes an interesting sentence, the meta-level goal statement

\[ \text{<- Demonstrate}(X',y), \text{Interesting}(y) \]

can be used, in theory at least, to generate interesting consequences of a given set of assumptions \( X \). Moreover, by solving the two problems cooperatively rather than sequentially, it is possible for the criteria characterising interesting sentences to guide the generation of consequences of \( X \).

The goal statement

\[ \text{<- Demonstrate}(t,Y') \]

where \( Y' \) names a given consequence and \( t \) is a partially instantiated term which names a given collection of assumptions \( X \) together with unknown additional assumptions \( x \), can be used to find the missing assumptions \( x \). The goal statement

\[ \text{<- Demonstrate}(t,Y_1'), \text{Demonstrate}(t,Y_2'), \ldots, \text{Demonstrate}(t,Y_m') \]

moreover, can be used to find missing assumptions which together with the given assumptions \( X \) imply all of the conclusions \( Y_1, Y_2, \ldots, Y_m \). In the simplest case, if the conclusions are sufficiently similar, the missing assumptions may be an inductive generalisation of the conclusions. Provided the proof procedure is sufficiently constrained it will avoid generating useless assumptions such as \( Y_1 \& Y_2 \& \ldots \& Y_m \), which trivially imply the conclusions.

Addition and suppression of assumptions

Languages in the PLANNER family and most versions of PROLOG achieve some of the power of the Demonstrate relation by providing facilities for adding and suppressing statements during the course of a demonstration. Instead of explicitly trying to solve a goal of the form

\[ \text{Demonstrate}(X',Y') \]

in these languages it is necessary to

- add the statements \( X \) to the program,
- try to show \( Y \), and then suppress \( X \) afterwards.

Since assumptions change dynamically during the course of a single demonstration, such programs can be exceedingly dangerous.

Addition and suppression of assumptions can be accomplished more safely by means of the Demonstrate relation. Moreover, efficiency can be achieved by directly executing the proof procedure recursively on the same machine or cooperatively on another machine instead of simulating it with the definition. On the other hand, Demonstrate goals of other
input-output patterns, which can not be solved by addition and suppression of assumptions, can be solved by using the definition. Addition and suppression of assumptions can only be used when the object language and meta-language are the same. But, provided the meta-language is sufficiently powerful, the Demonstrate relation can be used to connect any two languages.

**Bootstrapping**

The meta-language \( L_2 \) may differ in sophistication from the object language \( L_1 \). If it is less sophisticated to start with, then it can use its definition \( Pr \) of provability in \( L_1 \) to simulate \( L_1 \) and to increase its own sophistication. This is bootstrapping: the language \( L_2 \) pulling itself up by its own bootstraps, using the definition \( Pr \) to solve problems more intelligently than it would otherwise, acting the way it thinks a more intelligent proof procedure would behave.

Bootstrapping can be effective even if the more sophisticated language \( L_1 \) does not have an independent existence of its own. The definition, if it is consistent, can serve as a construction which causes the language \( L_1 \) to come into existence.

Bootstrapping, and more generally, defining an implementation of one language within another is a common technique in computing. An implementation of a language is created by writing a program which functions as a translator or interpreter for it in another existing language.

The clauses DL-2, which define the top-level of a Horn clause proof procedure \( L_1 \) can be used to bootstrap a simple top-down Horn clause proof procedure \( L_2 \) which executes procedure calls sequentially in the order in which they are written. By means of appropriate definitions of the rest of the program and of the procedure Select in particular, it is possible to define a proof procedure which executes procedure calls cooperatively. Although \( L_2 \) executes procedure calls sequentially, the new proof procedure \( L_1 \) executes procedure calls as coroutines according to the criteria specified in the procedure Select. By appropriate modification of the definition, other improvements, such as loop detection, intelligent backtracking and goal transformation, can also be incorporated in the new proof procedure \( L_1 \). More modestly, the definition of Demonstrate might only enhance the input syntax of \( L_2 \), defining infix notation for predicate symbols and function symbols, for example. More ambitiously, it might define a proof procedure for a richer version of logic, full clausal form or standard form, for example.

PROLOG systems and programs have used the bootstrapping technique since their first implementation in 1972 in Marseille. They have been used primarily for improving the input syntax and for corouting. A variety of Horn clause programs defining Horn clause provability have also been written at Imperial College. Simple Horn clause programs typically run about 100 times slower when simulated by using such definitions than they do when executed directly. PROLOG programs have also been written for non-Horn clause provability and by Broda for the standard form of logic. The PROLOG compiler written in PROLOG by Warren, Pereira, and Pereira [1977] and Colmerauer's [1977] interpreter.
for a restricted subset of natural language can also be regarded as applications of bootstrapping.

**Combining the object language and meta-language**

So far we have assumed an asymmetric relationship between the two languages $L_1$ and $L_2$. There is no reason in principle, however, why one language should know more about its companion than the other. Both languages might possess a definition of the other's proof procedure. Each language could serve as the other's meta-language and could simulate its proof procedure.

There is no reason either why the two languages should not be identical in all respects. It is possible therefore to have a single language equipped with a definition $Pr$ which is a correct representation of its own proof procedure. Given a problem of the form

$$\text{Demonstrate}(X', Y')$$

it can use the definition to simulate itself or equivalently it can show that

$$Y \text{ can be demonstrated from } X$$

directly. Solving the problem by direct execution is equivalent to the proof procedure calling itself recursively.

Such a relationship between object language and meta-language is already familiar in the programming language LISP [McCarthy et al 1962]. The function of a LISP interpreter or compiler is to evaluate an expression $y$ in an environment $x$, which defines the values of the symbols occurring in $y$, producing a result $z$ which is the value of $y$ in the environment $x$.

In functional notation this can be expressed

$$\text{eval}(x, y) = z,$$

which is like $\text{Demonstrate}$, except that the additional parameter $z$ names the output. We shall argue later that it is useful to extend $\text{Demonstrate}$ to a four argument relation

$$\text{Demonstrate}(x, y, u, z)$$

which holds when

- given the assumptions named $x$,
- the conclusion named $y$ and
- the control named $u$,
- the proof procedure generates the output named $z$.

The function $\text{eval}$ can be defined in LISP, like $\text{Demonstrate}$ can be defined in logic. In the same way that $\text{Demonstrate}$-goals with appropriate
input can be solved either by using the definition or by direct execution, eval-function calls can be evaluated in LISP either by using the definition of eval or by recursive invocation of the LISP evaluation mechanism. Since LISP functions have fixed input parameters, explicit use of the definition of eval can always be relaced by recursive invocation. Indeed, it was a study of the analogue in logic of eval in LISP which led the author and Ken Bowen to propose the amalgamation of object language and meta-language presented in this chapter.

**Incompleteness of the combined object and meta-language**

The combination of object language and meta-language avoids the paradoxes of self-reference in natural language. The attempt to reconstruct them leads instead to the construction of a true but unprovable sentence:

\[ D \rightarrow \neg \text{Demonstrate}(Pr', D) \]

which mentions its own name D. The term Pr' names the definition Pr of Demonstrate.

It is easy to show that, if Pr is consistent and correctly represents the provability relation, then neither the sentence named D nor its denial can be derived from Pr.

**Proof:**

Consider the two cases:

1. The sentence named D can be derived from Pr.
2. Its denial Demonstrate(Pr', D) can be derived from Pr.

**Case(1)** By the assumption of correct representability, (1) implies that

Demonstrate(Pr', D) can be derived from Pr. But then both the sentence and its denial can be derived from Pr, contradicting the assumption that Pr is consistent.

**Case(2)** By the assumption of correct representability (2) implies that

the sentence named D can be derived from Pr.

Again, both the sentence and its denial can be derived from Pr, contradicting the assumption that Pr is consistent.

Since both cases lead to contradiction, neither the sentence named D nor its denial can be derived from Pr.

But the proposition

The sentence named D can be derived from Pr.
or equivalently (by correct representability)

\[ \text{Demonstrate}(\text{Pr'}, D) \]

is either true or false of the provability relation. We have just shown (Case 1) it is not true. Therefore its denial

\[ D \quad \neg \text{Demonstrate}(\text{Pr'}, D) \]

is true, though unprovable.

The sentence named D is related to negation interpreted as failure. Given the problem

\[ \text{Demonstrate}(\text{Pr'}, D) \]

the proof procedure neither succeeds nor fails in finite time. (Finite failure would imply that

\[ D \quad \neg \text{Demonstrate}(\text{Pr'}, D) \]

could be proved from the iff-definition of Pr.) Thus the proof procedure does not terminate in its attempt to solve the problem, and therefore its denial

\[ D \quad \neg \text{Demonstrate}(\text{Pr'}, D) \]

truly states that the problem cannot be solved.

The sentence named D can be constructed in a variety of ways including the one used in Gödel's original incompleteness proof.

**More comprehensive form of the Demonstrate relation**

To simplify the discussion we have assumed that a proof procedure determines a two-place relation between assumptions and conclusions. In reality proof procedures are more complicated. They also accept control specifications which guide the proof strategy and they return output. It is more realistic, therefore, to regard a proof procedure as determining a four-place relation

\[ \text{Demonstrate}(x, y, u, z) \]

which holds when

- given the assumptions named \( x \),
- the conclusion named \( y \) and
- control named \( u \),
- the proof procedure generates the output named \( z \).

The control parameter \( u \) might specify, for example,

1. whether one proof method or another should be applied,
2. whether one, all or "best" solutions are required, and
3. whether a proof, trace of the search,
substitution for variables in the conclusion, or
simple Yes-No answer is required for the output z.

The trace of a proof procedure consists of the sequence of sentences
and other expressions generated by the proof procedure during the course
of searching for a solution. Thus the proof procedure may successfully
return as output the trace of an unsuccessful search for a solution. It
may also return a simple No-answer if it can determine that the search
space contains no solutions.

The more comprehensive form of the Demonstrate relation is useful for
obtaining and processing lists of all solutions. This is especially
useful in database applications to count all answers to a query or to
print the list of all answers as a table. Given a Horn clause database S
of suppliers and parts, for example, the Demonstrate relation can be used
both to formulate and answer the question

How many suppliers of stationery are located in London?

<- Demonstrate(S, atom(Supplies,X.Stationery,NIL),
       atom(Location,X.London,NIL),NIL, all(X),z),
       Count(z,w)).

Here all(X) specifies that a list of all distinct answers, consisting of
substitutions for the variable X, is required for the output z. Count(z,w)
can be defined by

      Count(NIL, 0) <-
      Count(u.v, w) <- Count(v,w'), Plus(w',1,w).

Instead of counting the list of all answers, a procedure

Format(z,w)

could rearrange the list z, inserting new page, new line and space
characters, so that the resulting list w, when printed, has the
appearance of a table.

Exercises

1) The top-level Dl-2 of the definition of the Horn clause provability
relation can be tested for the problem of the Fallible Greek without
defining the lower-level procedures in full. It suffices to supply
assertions which solve the sub-problems which arise during the course of
trying to solve the top-level problem. The following assertions are
sufficient for renaming the procedures Fl-4 and for finding the parts of
the resulting procedures.

    Renamevars(Fl, goals, Fl') <-
    Renamevars(F2, goals, F2 ) <-
    Renamevars(F3, goals, F3 ) <-
    Renamevars(F4, goals, F4 ) <-
    Parts(Fl', atom(Fallible,Y), atom(Human,Y),NIL) <-
Chapter 12: Formalisation of Provability

a) Supply assertions or simple procedures for the remaining conditions in Dl-2.

b) Using the assertions and simple procedures from (a), test Dl-2 for the problem of the Fallible Greek by using top-down inference and backtracking to find a solution.

2) Complete the definition Dl-2 of the Demonstrate relation by defining the lower-level procedures in full. For this purpose it is useful to employ a different data structure for naming expressions of the object language:

a) Predicate symbols and function symbols can be named by constant symbols.

b) Constant symbols can be named by terms const(t) where t names a number, e.g. 0, s(0), ... etc.

c) Variables can be named by terms var(t) where t names a number.

d) Composite terms can be named by terms of the form term(s, t) where s names a function symbol and t names a list of terms.

e) Atoms and lists of atoms in goal statements and procedure bodies can be named as before.

f) Procedures can be named by terms proc(s, t) where s names the head and t the body of the procedure.

g) Programs can be named by lists of the procedures they contain.

h) Substitutions can be named by lists of substitution components of the form sub(s, t) where s names a variable and t names a term.

Notice that a simple way to rename the variables in a procedure is to

i) find T the maximum t such that var(t) occurs in the goals and

ii) replace every occurrence of a variable var(s) in the procedure by an occurrence of the variable var(r) where r = s+T.

The simple definition of the Match relation

\[
\text{Match(expr1, expr2, sub)} \leftarrow \text{Apply(expr1, sub, expr3)},
\text{Apply(expr2, sub, expr3)}
\]

is liable to go into a loop when the two expressions do not match. A safer definition is the one which employs two substitution parameters, one for the current substitution which matches the parts of the two expressions which have been examined so far and another for the final
3) Modify the definition of the Demonstrate relation, defining the relationship

Demonstrate(prog, goals, sub)

which holds when the program solves the goals and generates a substitution of terms for the variables occurring in the goal as a solution.

This can be done at the top-level simply by adding extra conditions to D2. The substitution required in the head of the clause can be obtained by appropriately combining the substitution obtained by the recursive call to the Demonstrate procedure in the body of the clause together with the output component of the substitution which matches the selected goal with the head of the procedure.

4) Define the top-level of a deterministic Horn clause interpreter for Horn clause programs. The interpreter can be made deterministic by explicitly managing the search through the top-down search space one branch at a time.

Branches of a search space can be represented by lists of nodes. Each node consists of

i) the list of goals at the node,

ii) the selected goal, and

iii) the list of untried procedures which have not yet been applied to the goal.

To solve the initial collect of goals, process the branch whose only node consists of the initial goal statement, selected goal and the appropriate list of untried procedures.

Any program successfully processes a branch whose tip contains the empty list of goals.

To process a branch whose tip node contains a non-empty list of untried procedures for the selected goal try to match the goal with the head of the first untried procedure.

i) If the match fails, remove the procedure from the list of untried procedures and process the new branch.

ii) If the match succeeds, remove the procedure from the list of untried procedures, add a new tip containing the new goal statement obtained by applying the successful procedure, and process the new branch.

To process a branch whose tip node has an empty list of untried procedures for its selected goal, backtrack by deleting the tip from the branch and processing the new branch.
5) Show that for any set of clauses $S$ there exists a corresponding set of Horn clauses $S^*$ such that $S$ is consistent (or inconsistent) if and only if $S^*$ is. Thus any problem which can be expressed in clausal form can be expressed by means of Horn clauses using the correspondence $^*$. The correspondence can be established by showing that the provability relation for clauses in general can be defined by means of Horn clauses.