CHAPTER 8

The Connection Graph Proof Procedure

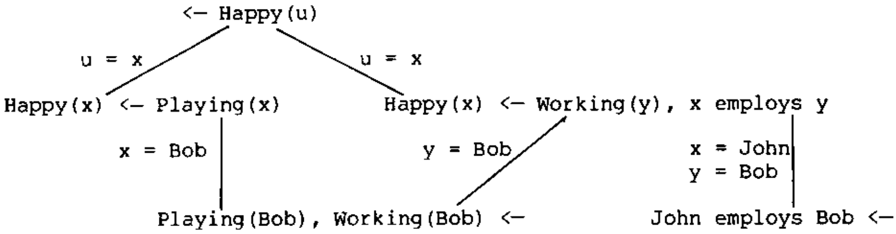
The search space determined by unrestricted application of the resolution rule is highly redundant. Redundancy can be avoided, at the expense of flexibility, by restricting resolution to top-down or bottom-up inference. It can also be avoided, however, without the loss of flexibility by employing the connection graph proof procedure.

Clauses are stored in a graph and occurrences of matching atoms on opposite sides of the arrow are connected by arcs. Associated with each arc in the graph is the resolvent obtained by resolving the clauses connected by the arc. The main operation of the connection graph proof procedure is the selection of an arbitrary arc and the incorporation of the associated resolvent into the connection graph. Top-down inference is performed by selecting an arc connected to a goal statement; bottom-up inference, by selecting an arc connected to a clause which contains no conditions. Redundancy is avoided by deleting the selected arc and by restricting the number of new arcs which are added when the resolvent is incorporated into the graph.

The initial connection graph

The first step of the connection graph proof procedure is the construction of the initial connection graph. In addition to the initial set of clauses, the initial connection graph contains an arc for every pair of matching atoms on opposite sides of the arrow in different clauses. The arc connects the atoms and is labelled by the matching substitution. Later in the chapter we consider the case in which an arc links atoms in the same clause.

The initial connection graph for a simple non-Horn clause problem is illustrated below.

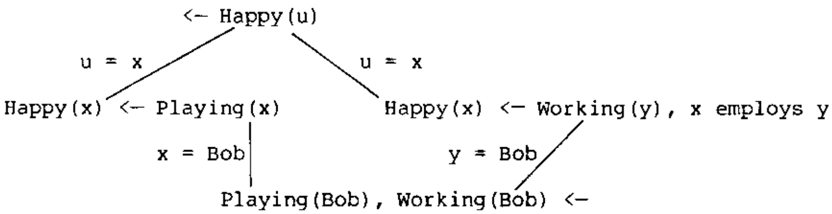


Associated with each arc in the graph is the resolvent obtained by matching the atoms linked by the arc. Conversely, for every resolvent which can be generated from different parent clauses there is an associated arc in the graph.

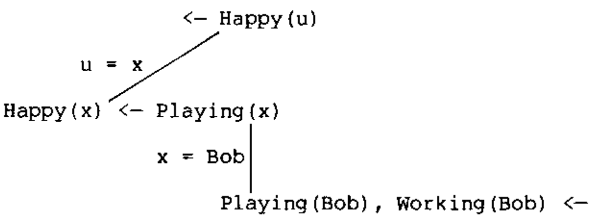
According to Robinson's purity principle [Robinson 1965a], a clause which contains an unlinked atom can be deleted from a set of clauses without affecting its consistency (or inconsistency). Such a clause can not contribute to a resolution refutation because the unlinked atom can not be resolved upon.

Deletion of clauses containing unlinked atoms is an important feature of the connection graph proof procedure. In addition to the clause itself, all links connected to its atoms must also be deleted from the graph. Deletion of such links, however, may cause atoms in other clauses to become unlinked. Thus deletion of clauses can create a chain reaction in which a succession of clauses is deleted from the graph. Deletion of clauses simplifies the connection graph, reduces the search space, and makes it easier to find a solution.

The effect of deleting clauses can be illustrated by assuming that Bob is unemployed and modifying the preceding example.



We delete the clause which contains the unlinked atom.

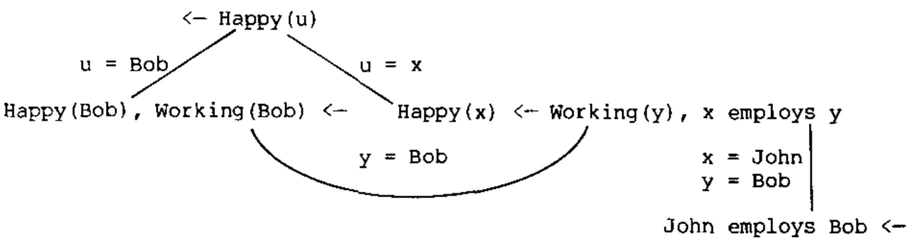


The new graph contains a new unlinked atom. Deletion of clauses continues until we are left with the empty set of clauses. The empty set of clauses is trivially consistent, because it contains no clauses which can be false in an interpretation. Therefore the original set of clauses is consistent as well.

The Resolution of links in connection graphs

The basic operation of the proof procedure is the selection of a link and the generation of the associated resolvent. The link is deleted and the resolvent is added to the graph. New links are added connecting atoms in the resolvent to atoms in the rest of the graph. The new links can be constructed, without searching the graph, from the links which are already connected to the atoms in the parent clauses.

For example, in the initial connection graph at the beginning of the chapter, we can reason bottom-up by selecting the link which matches the two atoms containing the predicate symbol Playing. In the resolvent, the atom Happy(Bob) descends from the atom Happy(x) in the parent clause. All new links connected to the new atom descend from the links connected to the parent atom. In this case the new link connecting Happy(Bob) to Happy(u) is derived from the old link connecting Happy(x) to Happy(u) . The new connection graph, which results from selecting the link, generating the resolvent, adding new links and deleting both parent clauses (which now contain unlinked atoms) is illustrated below.



The substitution u = Bob which labels the new link can be computed from the substitution x = Bob which labelled the selected link and the substitution u = x which labelled the "parent" link from which the new link descends.

Before continuing with the example we outline the definition of the proof procedure in general.

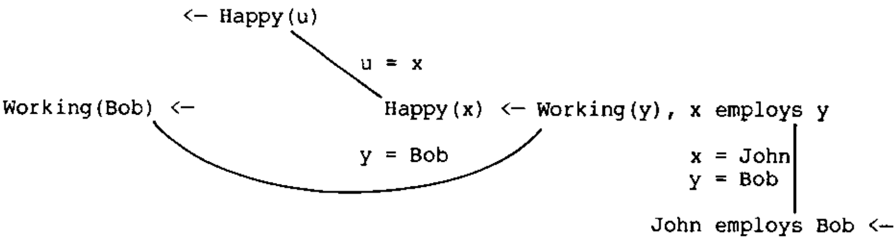
The *connection graph proof procedure* begins with an initial connection graph and processes it repeatedly until the empty clause is generated. It *processes a connection graph by*

(1) repeatedly deleting clauses containing unlinked atoms and deleting their associated links until all such clauses have been deleted and then

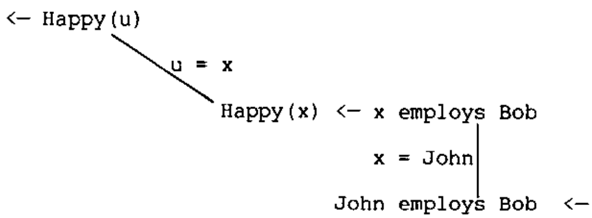
(2) selecting a link, deleting it and adding the resolvent and its associated new links to the graph.

This definition of the top-most level of the connection graph proof procedure is given in the "repeat-until" iterative style of algorithm description associated with Algol-like programming languages. At the end of the chapter, we shall reexpress the definition in the Horn clause logic programming style.

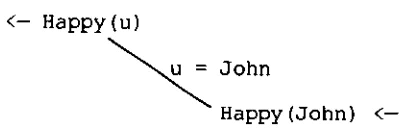
We return to the example. Any link may be selected from the graph. We shall continue, however, with the bottom-up analysis of the case Playing(Bob) by selecting the link labelled u =Bob. Deletion of the selected link leaves one of the parents with an unlinked atom. The parent is deleted.



The goal has now been solved in the first case Playing(Bob). Next we investigate the remaining case Working(Bob), also reasoning bottom-up. When the selected link is deleted, both parent clauses contain unlinked atoms and are deleted as well.



We continue to reason bottom-up and delete both parents because they contain unlinked atoms.



The resolvent associated with the remaining link is the empty clause and both parents are deleted.

□

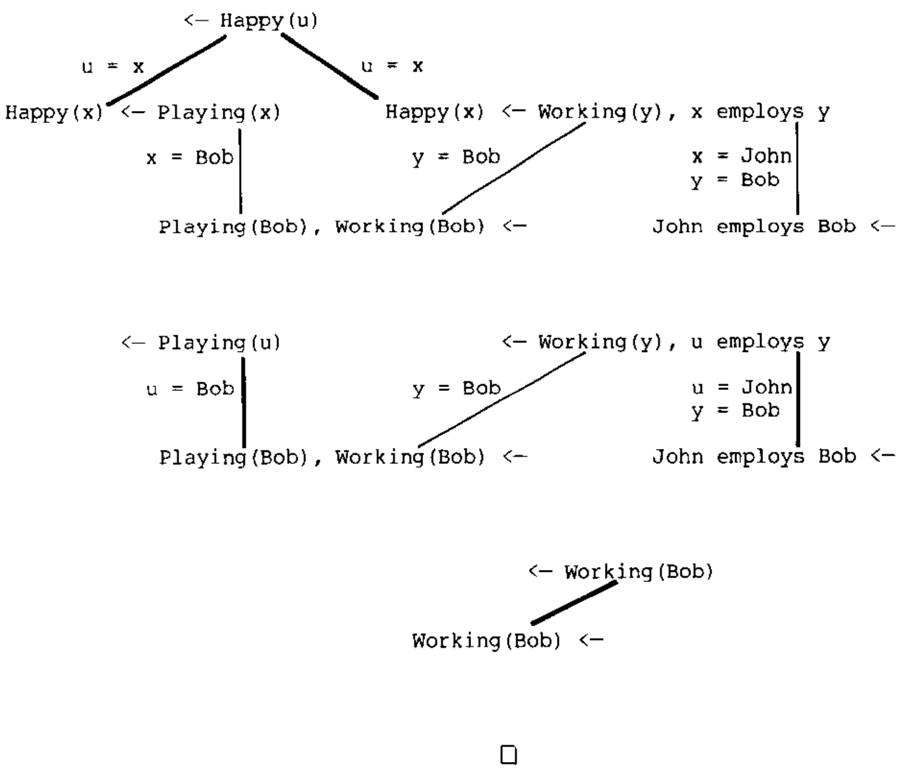
Notice that the proof gives a disjunctive answer to the question:

Is anyone happy?

Yes, Bob or John.

The sequence of successive connection graphs generated by the proof procedure constitutes both a proof of inconsistency as well as a search for the proof. In this example, every step in the search contributes to the proof itself. In the general case, however, according to a theorem of Ehrenfeucht and Rabin [Bundy 1971] [Meltzer 1972], it is not always possible to avoid steps which are not relevant to the proof.

At every stage during the course of searching for a proof, any link can be selected to generate a resolvent. The selection of different links leads to the generation of different search spaces, some of which may be easier to search than others. In the following sequence of connection graphs we illustrate a top-down search for a solution to the previous problem. Selected links are indicated by bold lines. Several links may be marked for selection in the same graph when the order of selection does not matter, in order to reduce the number of separate graphs displayed. Deletion of clauses containing unlinked atoms is not exhibited explicitly.

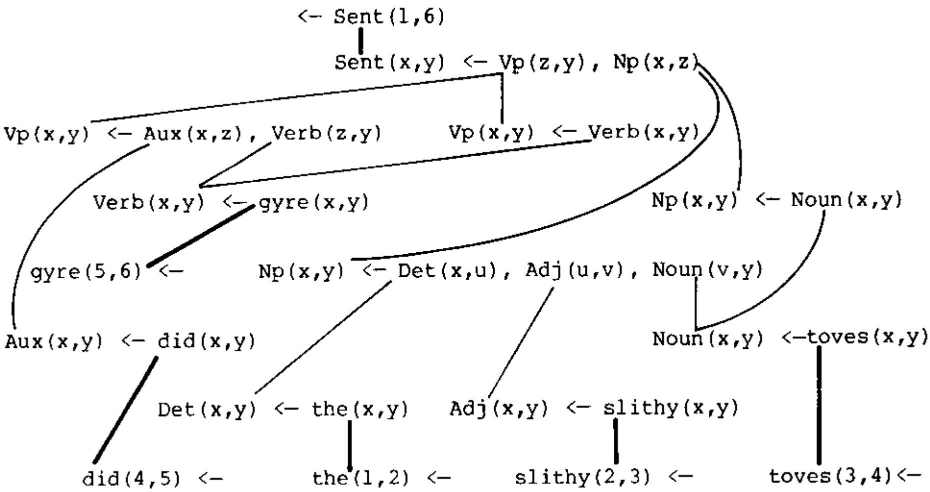


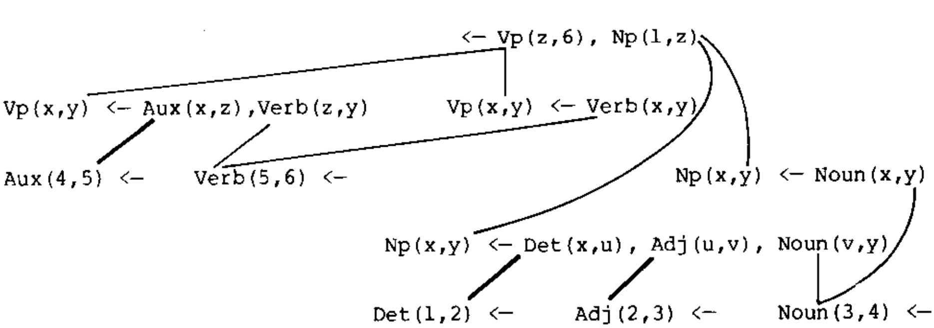
As in the bottom-up search for a solution, every step contributes to the proof.

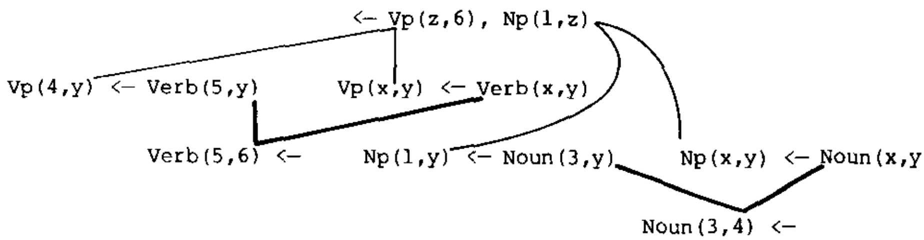
Notice that unrestricted application of the resolution rule is redundant in the sense that it determines a search space which contains many unnecessary clauses including, in particular, all those which belong to both the top-down and the bottom-up search spaces exhibited above.

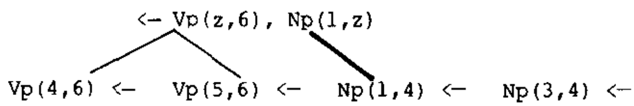
Mixed top-down and bottom-up search - the parsing problem

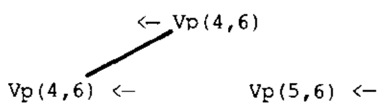
Top-down and bottom-up inference can be mixed, simply by mixing the selection of links connected to atoms in goal statements with the selection of links connected to atoms in clauses which contain no conditions. In general it is useful always to select a link which results in the least complicated new graph. This strategy applied to a version of the parsing problem of Chapter 3 results in a mixed top-down, bottom-up search. As in the preceding example, selected links are indicated by bold lines. Substitutions, which label links, are omitted from the graph.











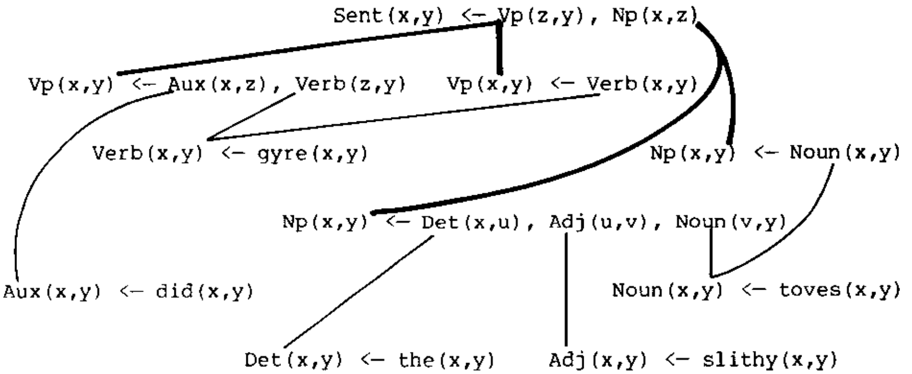
□

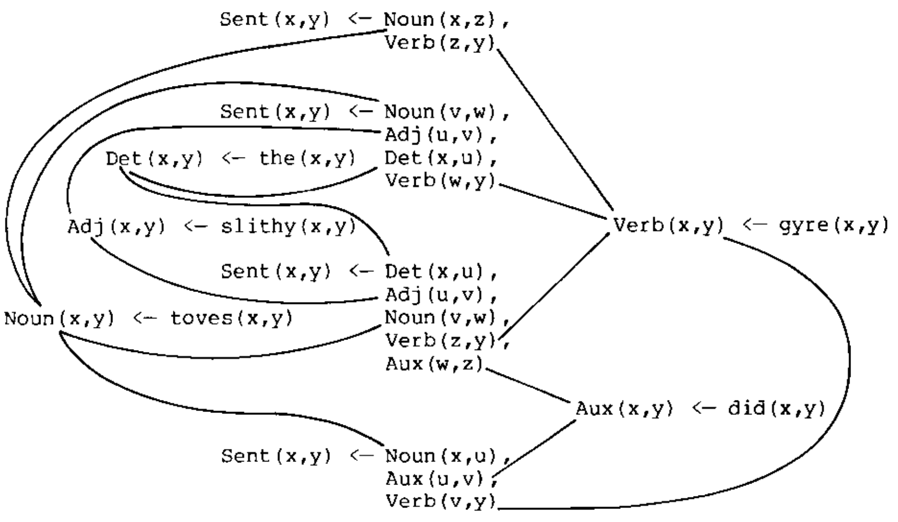
Macro-processing and middle-out reasoning

In conventional programming languages, macro-processing transforms a program by eliminating all calls to a given procedure, executing them in advance of the particular problems to be solved. The original procedures are replaced by the new ones. The analogue of macro-processing in logic is middle-out reasoning combined with deletion of the parent clauses because they contain unlinked atoms.

Macro-processing has the advantage that procedure calls are executed once and for all before the problems are given, rather than repeatedly during the course of trying to solve them.

Macro-processing can be illustrated by eliminating all calls to the Np and Vp procedures in the parsing problem.

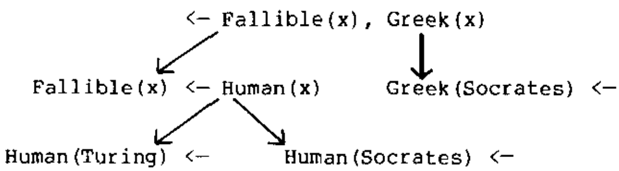


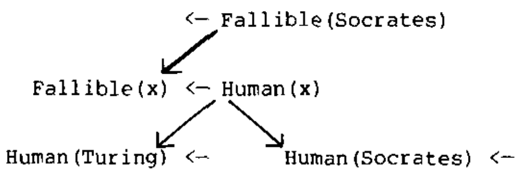


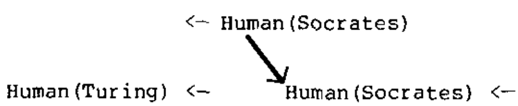
Arrow notation for controlling selection of links

The arrow notation, introduced informally earlier in the book, can be used to control the selection of links in the connection graph proof procedure. The links of a connection graph can be turned into arrows by giving them a direction. A clause is regarded as active if all links connected to its atoms are outgoing. A link may be selected if it is connected to an atom in an active clause. The new links connected to atoms in a resolvent inherit their direction from the parent links from which they descend.

The connection graph proof procedure can be restricted to top-down inference, by directing all arrows from conditions to conclusions. Then a clause is active if and only if it is a goal statement. The following sequence of graphs illustrates the use of arrow notation to impose a top-down problem-solving interpretation on the problem of the fallible Greek. Despite notational similarities, there is no connection between arrow notation in connection graphs and arcs in semantic networks.

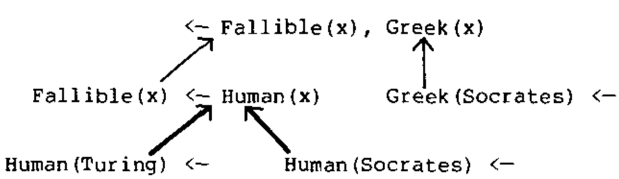


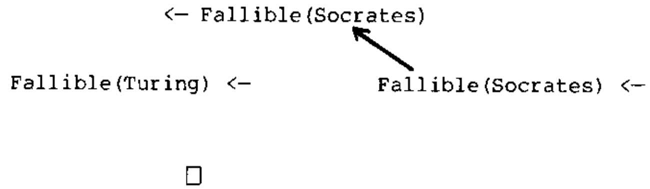




□

The proof procedure can be restricted to bottom-up inference, by directing all arrows from conclusions to conditions. Then a clause is active if and only if it has no conditions. The use of arrow notation for bottom-up inference is illustrated below.





The arrow notation can be used with non-Horn clauses to control the generation of assertions for use in the solution of subgoals. The non-Horn clause in the connection graph below, for example, generates the assertion

Studentof(☺,Bob) <-

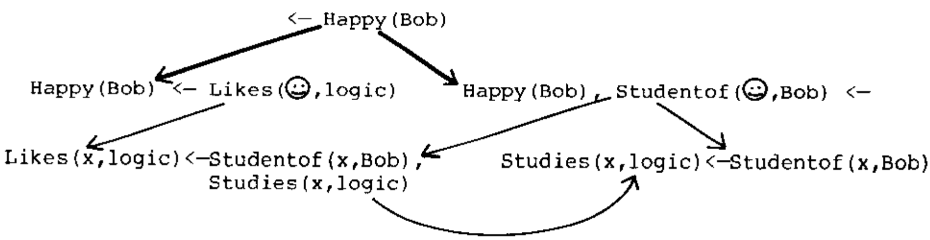
to assist the solution of the subgoal

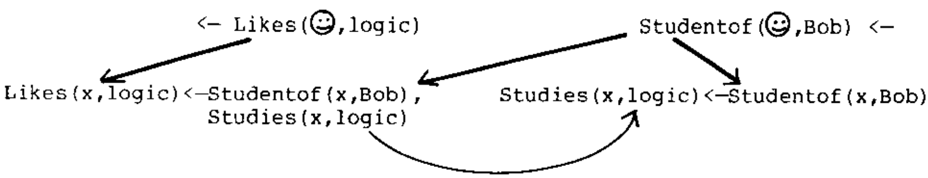
<- Likes(☺,logic).

The two clauses from which the assertion and subgoal are derived, together with the associated arrow notation, attempt to show that Bob is happy by asserting that is a student of Bob and showing that even likes logic. Since nothing else is said about the individual , if it can be shown that likes logic, then anyone who is a student of Bob likes logic. The two clauses, therefore, state in effect that

Bob is happy if all his students like logic.

The arrows in the following connection graph direct the search for a solution top-down from the top-level goal and the derived subgoal, but bottom-up from the assertion to be used in solving the subgoal.





<- Studies(☺,logic)   Studies(☺,logic) <-

□

Notice that Bob would also be happy if he had no students

<- Studentof(x,Bob)

or if everyone liked logic unconditionally

Likes(x,logic) <- .

There is no guarantee that every assignment of direction preserves the solvability of a connection graph. It seems sensible, moreover, to restrict the direction of arrows so that all links connected to the same atom have the same direction.

Self-resolving clauses

A self-resolving clause is one which resolves with a copy of itself. For example, the clause

Append(x.y, z, x.y') <- Append(y,z,y')

resolves with the copy

Append(u.v, w, u.v') <- Append(v,w,v').

For the sake of completeness, it is necessary to connect resolving atoms in a self-resolving clause by means of a link.

Append(x.y, z, x.y') <- Append(y,z,y')

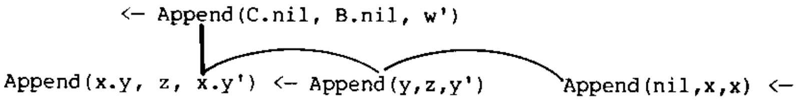
Such a link is a pseudo-link in the sense that is stands for a link between atoms in different copies of the same clause.

Pseudo-links can be selected for processing, but it is simpler for the purposes of exposition to restrict their use to the derivation of new links. This is illustrated in the following example.

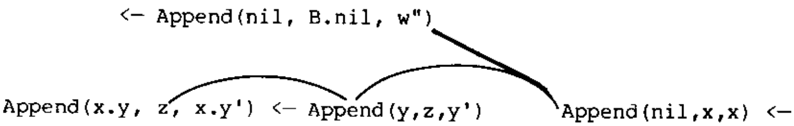
<- Append(A.C.nil, B.nil, w)

Append(x.y, z, x.y') <- Append(y,z,y')   Append(nil,x,x) <-

The single atom in the resolvent descends from an atom having two links, one of which is a pseudo-link. The pseudo-link gives rise to a descendant which is a normal link. The other link connected to the assertion has no descendant. The original goal statement contains an unlinked atom and therefore is discarded when the resolvent is added to the graph.



The new graph is similar to the initial connection graph. However, this time, when the resolvent is generated, it is the pseudo-link which has no descendant and the link to the assertion which has.



The resolvent of the new link is the empty clause. Independently, the recursive clause can be deleted because its conclusion has only a pseudo-link. Once the recursive clause has been deleted, the assertion can be deleted as well. The resulting connection graph consists of the empty clause alone.

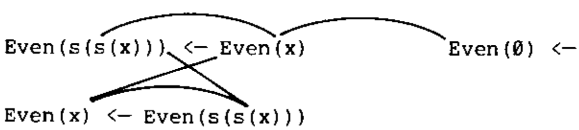
□

In general, a self-resolving clause can be deleted if one of its atoms has no normal (non-pseudo-) links. The inheritance of links and pseudo-links in connection graphs has been studied by Bruynooghe [1977]. Note that, although in all of the preceeding examples the final connection graph contains only the empty clause, in the general case it may contain other clauses as well.

Deletion of links whose resolvents are tautologies

A clause is a *tautology* if it contains the same atom both as a condition and as a conclusion. The use of tautologies in top-down problem-solving leads to loops in which a goal reoccurs as its own subgoal. For that reason, because they do not positively contribute to the solution of problems, tautologies can be deleted from a set of clauses without affecting inconsistency [Robinson 1965a]. In the connection graph proof procedure, the effect of deleting tautologies can be obtained by deleting links whose resolvents are tautologies.

The set of clauses describing the concept of even number is an example.



The two links connecting the two recursive clauses have resolvents which are tautologies. The links are deleted from the graph:

Even(s(s(x))) <- Even(x)   Even(Ø) <-

Even(x) <- Even(s(s(x)))

The collection of three clauses is consistent because it contains no goal statement. The two recursive clauses can be deleted because they contain atoms with only pseudo-links. The basis assertion can then be deleted as well. Given the goal statement

<- Even(s(s(s(s(Ø)))))

moreover, the condition of the second recursive clause still has no non-pseudo-link. Consequently, the clause can be deleted, leaving the simpler graph:

<- Even(s(s(s(s(Ø)))))

Even(s(s(x))) <- Even(x)   Even(Ø) <-

In more complex examples it is not so easy to recognise that a clause cannot contribute to a solution. In such cases a more global analysis may be useful. Global problem-solving strategies are investigated in the next chapter.

The connection graph proof procedure

We summarise here the definition of the connection graph proof procedure in a style of English which corresponds to the procedural interpretation of Horn clauses.

To demonstrate the inconsistency of a set of clauses by the connection graph proof procedure, generate and solve its initial connection graph.

The initial connection graph for a set of clauses contains all clauses in the set, a (non-pseudo-) link connecting each pair of matching atoms on opposite sides of the arrow in different clauses, and a pseudo-link connecting atoms on opposite sides of the arrow in the same clause if the atoms match in different copies of the clause.

A connection graph is solved if it contains the empty clause.

To solve a connection graph which does not contain the empty clause,

either delete a link whose resolvent is a tautology, and solve the resulting connection graph,
or delete a clause containing an unlinked atom, together with its associated links, and solve the resulting connection graph,
or select a link which is not a pseudo-link, delete it, add the resolvent together with its new links to the graph, and solve the resulting connection graph.

A (*non-pseudo-*) *link* connects an occurrence L of an atom in a resolvent to an occurrence K of an atom in another clause if L and K match, L descends from an occurrence L' of an atom in a parent clause, and there is a link (possibly a pseudo-link) between L' and K.

parent L' C' K   or   parent C' L' K D
resolvent L C   resolvent C L

A *pseudo-link* connects L and K in a resolvent if L and K match, L and K descend from L' and K' in the (same or different) parent clauses, and there is a link between L' and K'.

parent L' C' K'   or   parents C' L' K' D
resolvent L C K   resolvent L C K

The four different ways of solving a connection graph correspond to four clauses having the same conclusion. Ignoring the deletion of links whose resolvents are tautologies, the resulting three procedures express the logic and top-down control of the iterative algorithm described at the beginning of the chapter. The earlier algorithm can be obtained from the new one by further specifying the control over the use of the procedures given here. In particular,

(1) the alternative ways of solving a connection graph should be tried one at a time in the order in which they are written above and

(2) backtracking should not be employed, as the non-determinism$_1$ of the procedures doesn't matter.

The proof procedure which has been described is incomplete as it stands, because the factoring operation has been omitted. In order to avoid redundancy, severe restrictions need to be imposed on its use. Since adequate restrictions have not yet been devised, and since it simplifies the description of the proof procedure, we have decided to ignore the factoring operation altogether. A definition of the proof procedure including factoring can be found in the original publication [Kowalski 1974a].

The completeness of the connection graph proof procedure cannot be assured if the selection of links which are needed for a proof is postponed indefinitely. Such indefinite postponement might arise, for example, when the selection strategy carries out a depth-first search along a non-terminating path of a top-down search space. The requirement that every link eventually be scheduled for selection is the analogue of the exhaustiveness of search strategies for more conventional proof procedures.

A completeness proof for a variant of the connection graph proof procedure has been constructed by Brown [unpublished]. In the case of Horn clauses, his proof applies also to the proof procedure which has been described here. Other completeness proofs for the general case have been announced by Siekmann and Stephan [1976] and by Bibel [1979].

A number of proof procedures employ connection graphs but process them in a manner different from the one described here. Noteworthy among these are those of Sickel [1976] and Kellogg, Klahr and Travis [1978]. Closer to the connection graph procedure, however, is the unpublished cancellation system of Colmerauer.

Exercises

1) Express the top-level of the definition of the connection graph proof procedure by means of Horn clauses.

2) Using the methods described later in Chapter 10 for transforming sentences from the standard form of logic into clausal form, the definition of subset can be expressed by means of the following two clauses:

$$x \subseteq y, \quad arb(x,y) \in x \leftarrow$$
$$x \subseteq y \leftarrow arb(x,y) \in y$$

Used top-down these clauses behave as procedures which given a subgoal of the form $x \subseteq y$,

> assert that some arbitry individual, say arb(x,y), belongs to x and try to show that it belongs to y.

Use the connection graph proof procedure to prove the following theorems.

a) The empty set $\emptyset$ defined by

$$\leftarrow x \in \emptyset$$

is a subset of any set S.

b) Every set S is a subset of the universal set defined by

$$x \in U \leftarrow$$

c) Every set is a subset of itself.

d) The set A such that

```
a(x), b(x) <- x ∈ A
```

is a subset of the set B such that

```
x ∈ B <- a(x)
x ∈ B <- b(x)
x ∈ B <- c(x).
```

This is a formulation without equality of the problem of showing that

$\{a,b\} \subseteq \{a,b,c,\}$.

3) Verify the claim made in Chapter 5 that, using the connection graph proof procedure, bottom-up execution of the definition

```
Fib(0, s(0)) <-
Fib(s(s(x)), w) <- Fib(s(x), u), Fib(x,v), Plus(u,v,w)
```

of Fibonacci number requires only a constant amount of storage. Assume that the Plus relation is defined by means of variable-free assertions and ignore the space that would be needed to store them.