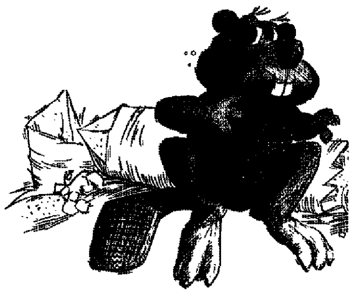

The Logical Definition of Deduction Systems

David L. Poole

Logic Programming Group
Research Report
CS-84-12

June, 1984

The Logical Definition of Deduction Systems

David L Poole

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

ABSTRACT

The separation of an algorithm into logic + control has special benefits in the definition of problem solving systems. Such a separation allows proving the correctness and completeness of the logic base, independent of the control imposed. The control component can then be developed to enhance the efficiency and explanation whilst preserving the logical power and correctness of the system.

This paper presents the definition of one such logical base for the non clausal first order predicate calculus. This logical base specifies how to transform a predicate calculus deduction problem into a problem of searching an and/or tree. Different control strategies are developed, with resulting systems combining the efficiency of connection graph proof procedures with the efficiency of non-chronological backtracking. Each implementation uses the input form of the unconstrained first order predicate calculus, with each step being locally explainable in terms of input given by the user. This allows the debugging and explanation facilities of expert systems to be incorporated into the implementations.

The logic can be extended to include bidirectional search and more powerful logics, for example the use of defaults. For each of these logical definitions different control strategies can be used, forming a family of logically equivalent, but heuristically distinct problem solvers. This separation, together with the use of the input form, forms a powerful vehicle for the systematic study of domain specific control strategies.

1. Introduction

1.1. Motivation

The separation of programs into (supposedly orthogonal) Logic + Control (Kowalski[79]) or epistemological and heuristic (McCarthy and Hayes[69]) components has been advocated by a number of people (eg Georgeff[82]). One of the most promising areas for such a separation is in the definition of problem solvers (also called automated reasoning systems, theorem provers, expert systems etc), where the logic part defines the semantics of the problem solver. Its power, correctness and completeness can be shown independent of any control strategies. The control part can be varied to produce various domain specific strategies, for semantically equivalent, but heuristicly distinct problem solvers. The logic component can be varied, virtually independent of the control strategy to include such things as bidirectional search, default logic (Poole[84]), or more exotic logics (eg actions, modal, relevance). Such a separation gives us a mechanism to understand the differences and similarities between various problem solvers.

1.2. The Approach

According to such a thesis, to implement some system, one axiomatises the relation to be implemented (the logic) and then separately define the control. The relation that we want to define is the relation that something can be proven from the facts given.

Here we define such a relation between components of the facts given by the user. We axiomatise the relation as a set of horn clauses. This axiomatisation is shown to be correct and complete with respect to the semantics given to the relation. A number of different ways to implement problem solvers from the axioms will be discussed. A comparison with other state of the art deduction systems will be given, showing how the resulting systems improves on each of them. The relation will first be defined and axiomatised for the propositional calculus and then extended to the predicate calculus.

The axioms should not be seen as rules of inference in themselves. The axioms are the logical definition of the PROVEN relation. We show that this definition correctly and completely characterises the desired semantics of logical derivability. These axioms together with some control component (search strategy), can indeed form the rules of inference for some complete problem solver.

2. Propositional Calculus

We assume that the user gives the system a number of well formed formulae (wffs), which we will call the **facts.** We will assume, without loss of generality, that there is one goal, namely the atom *success*. If the user has a goal G then she will add the fact $G \supset success$.

Define an **input wff** to be a fact or a well formed subformula of a fact. Distinct instances being distinct input wffs. There is also an implicit input wff, representing the atom *success* which will be the top level goal.

Each input wff can be associated with its principle operator or with the atom that it represents if there is no operator. Each atom and operator has its associated input wff. Input wffs are isomorphic to nodes in the parse tree of the facts, given the normal definition of a wff.

Example 1: The following fact has names associated with each input wff.

$$\begin{array}{ccccccccc} ((\neg & A & \wedge & B) & \vee & (E & \wedge & \neg & B)) \\ w_3 & w_4 & w_2 & w_5 & w_1 & w_7 & w_6 & w_8 & w_9 \end{array}$$

The following are the named input wffs of the fact:

w_1: $((\neg A \wedge B) \vee (E \wedge \neg B))$;
w_2: $(\neg A \wedge B)$;
w_3: $\neg A$;
w_4: A;
w_5: B;
w_6: $(E \wedge \neg B)$;
w_7: E;
w_8: $\neg B$.
w_9: B.

Note that w_5 and w_9 are different input wffs representing the same wff.

Definition: a **signed input wff** is a pair $<s,w>$ where $s \in \{p,n\}$ is called a sign; and w is an input wff.

Define the **value** $<s,w>^*$ of a signed input wff $<s,w>$ to be the wff w if $s=$ "p" and the wff $(\neg w)$ if $s=$ "n".

The definition of the value formalises the intuitive notion that "p" represents the positive of the wff, and "n", the negative of the wff.

Define the operator "$\sim$" on signs by

$$\sim n = p; \ \ \sim p = n$$

Lemma: $<\sim s,w>^* = \neg <s,w>^*$ -- proof follows from definitions.

We now define the relation on input wffs that we want to axiomatise. The relation is of the form:

$$PROVEN(\alpha,C)$$

where α is a signed input wff;
and C is a set of signed input wffs, called the **case set**.

Informally this is meant to mean that given the cases specified by C, α is proven.

More formally, the semantics that we want to give the relation will be:

$$facts \vdash C^* \supset w^*$$

where $C^* = \bigwedge_{\gamma \in C} \gamma^*$

Before we axiomatise the relation let us give an example of the sort of reasoning that we want to axiomatise.

Example 2: Consider the fact given in example 1, with the same naming of input wffs. Suppose w_0 is an input wff representing the atom "A", and w_9 is an input wff representing the atom "E", and furthermore suppose that w_9 has been proven given some case set "C". That is $PROVEN(<p,w_0>,C)$ has been derived. Then the following reasoning can be used:

(1) A is proven -- $PROVEN(<p,w_4>,C)$
(2) from (1) $\neg A$ must be proven false -- $PROVEN(<n,w_3>,C)$
(3) from (2) $(\neg A \wedge B)$ must be false -- $PROVEN(<n,w_2>,C)$
(4) as it is a fact, $((\neg A \wedge B) \vee (E \wedge \neg B))$ must be proven -- $PROVEN(<p,w_1>,C)$
(5) from (3) and (4) $(E \wedge \neg B)$ must be true -- $PROVEN(<p,w_6>,C)$
(6) from (5) E must be true -- $PROVEN(<p,w_7>,C)$
(7) from (6) the other E must be true -- $PROVEN(<p,w_9>,C)$

2.1. Axiomatising the PROVEN Relation

The proven relation can be axiomatised as follows.

1. Equality Substitution Rule

If w and y are input wffs representing the same atom then

$$\text{if } PROVEN(<s,w>,C) \text{ then } PROVEN(<s,y>,C)$$

The validity of this is obvious from noting that as far as the logic is concerned w and y are identical.

2. Facts Rule

If f is a fact ($f \in facts$) then

$$PROVEN(<p,f>,C)$$

This says that if f is a fact then it is proven no matter what C is.

3. Negation Rules

If w is of the form $(\neg z)$ then

3(a) Upward Negation Rule

$$\text{if } PROVEN(<s,z>,C) \textit{ then } PROVEN(<\sim s,w>)$$

3(b) Downward Negation Rule

$$\text{if } PROVEN(<s,w>,C) \textit{ then } PROVEN(<\sim s,z>,C)$$

The correctness of these rules follows directly from the semantics of *PROVEN* and the meaning of "~".

4. PROVEN for Binary Logical Connectives

The only other form an input wff w can have is $(w_1 \ op \ w_2)$ where "op" is one of $\{\wedge,\vee,\supset,\equiv\}$

The three operators "$\wedge$", "$\vee$" and "$\supset$" can be grouped together, as they all can be characterised by the use of "not" and "or". Note that $(w_1 \wedge w_2) \equiv \neg(\neg w_1 \vee \neg w_2)$ and $(w_1 \supset w_2) \equiv \neg w_1 \vee w_2$.

For each operator $op \in \{\wedge,\vee,\supset\}$ define op_i for $i \in \{0,1,2\}$ by $w = (w_1 \ op \ w_2)$ iff

$$<op_0,w>^* = <op_1,w_1>^* \vee <op_2,w_2>^*$$

$$\begin{aligned}
w &= (w_1 \vee w_2) \text{ iff } <p,w>^* = <p,w_1>^* \vee <p,w_2>^* \\
w &= (w_1 \wedge w_2) \text{ iff } <n,w>^* = <n,w_1>^* \vee <n,w_2>^* \\
w &= (w_1 \supset w_2) \text{ iff } <p,w>^* = <n,w_1>^* \vee <p,w_2>^*
\end{aligned}$$

Thus for $op = \vee$, $op_i =$ 'p' for all i; for $op = \wedge$, $op_i =$ 'n' for all i; and for $op = \supset$, $op_0 = op_2 =$ 'p';$op_1 =$ 'n'. The values of op_i uniquely define the operator.

If w is of the form $(w_1 \ op \ w_2)$ then the following rules apply

4(a) Upward Disjunction Rule

for $j \in \{1,2\}$,

$$\text{if } PROVEN(<op_j,w_j>,C) \textit{ then } PROVEN(<op_0,w>,C)$$

This rule may be paraphrased as "If one component of a disjunction is proven then the disjunction is proven" or "If one component of a conjunction is proven false then the conjunction is proven false" or "If the left hand side of an implication is proven false, or the right hand side is proven true then the implication is proven true". Each of these meanings is derived from the abstraction of "*op*".

Proof of Correctness:
Let $\{i,j\}=\{1,2\}$
$PROVEN(<op_j,w_j>,C)$
means: $facts \vdash C^* \supset <op_j,w_j>^*$
so then by disjunction introduction: $facts \vdash C^* \supset <op_j,w_j>^* \vee <op_i,w_i>^*$
so by definition above: $facts \vdash C^* \supset <op_0,w>^*$
that is: $PROVEN(<op_0,w>,C)$
Q.E.D.

The proofs of the other rules for binary operators are as straightforward as this, and will be left as an exercise to the reader.

4(b) Upward Conjunction Rule

$$\text{if } PROVEN(<\sim op_1,w_1>,C) \textit{ and } PROVEN(<\sim op_2,w_2>,C)$$
$$\textit{then } PROVEN(<\sim op_0,w>,C)$$

This may be paraphrased as "If both elements of a disjunction are proven false then the disjunction is proven false" or "If both sides of a conjunction are proven true then the conjunction is proven true" or "If the left hand side of an implication is true and the right hand side is false then the implication is false".

4(c) Downward Conjunction Rule

$$\text{for } j \in \{1,2\}, \text{ if } PROVEN(<\sim op_0,w>,C) \textit{ then } PROVEN(<\sim op_j,w_j>,C)$$

Which may be paraphrased as "If a conjunction is proven true then each conjunct is proven true", and similarly for disjunction and implication.

4(d) Implication Rule

$$\text{for } \{j,k\}=\{1,2\}, \text{ if } PROVEN(<op_0,w>,C) andPROVEN(<\sim op_j,w_j>,C)$$
$$\textit{then } PROVEN(<op_k,s_k>,C)$$

This may be paraphrased as "If an implication is proven and the left hand side is proven, then the right hand side must be proven", and similarly for the contrapositive implication, for disjunction and for conjunction.

4(e) Equivalence Rule

If w_0 is of the form $(w_1 \equiv w_2)$ and $\{i,j,k\}=\{0,1,2\}$ and
$PROVEN(<s_i,w_i>,C)$ and $PROVEN(<s_j,w_j>,C)$ then $PROVEN(<s_k,w_k>,C)$
where $s_k = \begin{cases} \text{"}p\text{"} \; if \; s_i = s_j \\ \text{"}n\text{"} \; if \; s_i \neq s_j \end{cases}$

This follows from noticing that if any two of the three input wffs involved in an equivalence are both proven true or both proven false then the third component is proven true, but if one is proven true and the other proven false then the third component is proven false.

Use of The Case Set

The preceding rules are not powerful enough to competely characterise the desired relation. There needs to be a mechanism to do what is commonly called *case analysis*.

For example, using only the rules above, *success* cannot be proven from the facts:

$$\underset{w_1}{A}\ \underset{w_2}{\supset}\ \underset{w_3}{success}$$

$$\underset{w_4}{B}\ \underset{w_5}{\supset}\ \underset{w_6}{success}$$

$$\underset{w_7}{A}\ \underset{w_8}{\vee}\ \underset{w_9}{B}$$

Two rules will be defined to use the case set to allow such reasoning to complete the axiomatisation.

5(a) Assuming Rule

$$PROVEN(\alpha, C \cup \{\alpha\})$$

This may be paraphrased as "α^* may be assumed by adding it to the case set". It means that "α^* is proven given the case that (amongst other things) α^*". Correctness follows directly from the desired semantics of PROVEN.

5(b) The Unassuming Rule

$$\textit{If } PROVEN(\langle s,w\rangle, C \cup \{\langle \sim s,w\rangle\}) \textit{ then } PROVEN(\langle s,w\rangle, C)$$

This may be paraphrased as "If we consider the case of $\langle \sim s,w\rangle^*$ and subsequently prove $\langle s,w\rangle^*$ then $\langle s,w\rangle^*$ must be true, not depending on $\langle \sim s,w\rangle^*$". This is saying no more than "If $A \wedge \neg z \supset z$ then $A \supset z$", where z is $\langle s,w\rangle^*$, which can easily be seen to be correct. This is an instance of the inference rule *reductio ad absurdum*.

2.1.1. An Example

This example shows a simple case of how the case set can be used. We will prove the input wff $w_{success}$ representing the wff *success* follows from the facts given above.

1. $PROVEN(<n,w_{success}>,\{<n,w_{success}>\})$ -- rule 5(a)
2. $PROVEN(<n,w_3>,\{<n,w_{success}>\})$ -- rule 1, step 1
3. $PROVEN(<p,w_2>,\{<n,w_{success}>\})$ -- rule 2
4. $PROVEN(<n,w_1>,\{<n,w_{success}>\})$ -- rule 4(d), steps 2,3
5. $PROVEN(<n,w_7>,\{<n,w_{success}>\})$ -- rule 1, step 4
6. $PROVEN(<p,w_8>,\{<n,w_{success}>\})$ -- rule 2
7. $PROVEN(<p,w_9>,\{<n,w_{success}>\})$ -- rule 4(d), steps 5,6
8. $PROVEN(<p,w_4>,\{<n,w_{success}>\})$ -- rule 1, step 7
9. $PROVEN(<p,w_5>,\{<n,w_{success}>\})$ -- rule 2
10. $PROVEN(<p,w_6>,\{<n,w_{success}>\})$ -- rule 4(d), steps 8,9
11. $PROVEN(<p,w_{success}>,\{<n,w_{success}>\})$ -- rule 1, step 10
12. $PROVEN(<p,w_{success}>,\{\})$ -- rule 5b, step 11

2.2. Status of the Axiomatisation

The preceding axiomatisation can be proven correct with respect to the semantics given for *PROVEN* (Poole[82]). That is it models the relation:

$$facts \vdash C^* \supset w^*$$

This can also be proven complete (Poole[82]), in the sense that if the facts are consistent and $facts \vdash success$, then

$$PROVEN(<p,w_{success}>,\{\})$$

can be deduced using the axioms (where $w_{success}$ is the implicit input wff representing the atom *success*).

2.3. Use of the Axioms

The first thing to notice about the axioms is that they are a set of horn clauses. So any search method for horn clauses can be used as a control component. In particular we can forward chain on the rules or backward chain.

Forward chaining will be similar to the so called ***natural deduction*** and will use the rules as they have been justified above. The main problem with forward chaining is in knowing when to use the rules, and in particular when to use the ***assuming rule.***

Backward chaining on the rules corresponds to a goal directed search. The most important thing to notice is that the ***unassuming axiom*** is applicable at every stage in the proof; that using it twice in a row does not produce a new subgoal; and that using the rule at any time will not prevent any other rule from being applicable, or preclude any solution that will be applicable without using the rule. Having the maximal possible case set will enable the assuming rule to be used whenever possible. When backward chaining the ***unassuming rule*** should take its unassuming role in the background, adding elements to the case set at each step.

The use of the ***unassuming axiom*** in this way corresponds directly to the framing of literals in linear resolution (Chang and Lee[73]), and the use of the ***assuming axiom*** corresponds to the reduction of a reducible ordered clause. In fact one possible control strategy to use is a backtracking right to left search which is linear resolution, except it is not restricted to clause form.

3. The Predicate Calculus

As promised before we will now extend the relation and the axiomatisation to the predicate calculus. Assume that the facts are skolemised. (It is possible to define rules for the quantifiers, but that will only complicate the presentation, the only thing lost by skolemising is the need to expand equivalences which contain quantifiers.)

To extend the relation to the predicate calculus we basically add a substitution to the relation. The problem that arises with such a simple minded approach is in the need to rename different variables, and to copy wffs that need to have more than one instance in a proof. To overcome both of these problems we introduce the notion of indexed variables.

Definition: An **indexed variable** is a pair, written v^i such that v is a variable in the object language (i.e. what the user types in), and i is an integer, called the index of the indexed variable. Two indexed variables are the same if and only if they have identical components.

If w is an input wff, and i is an integer then w^i is called an **indexed wff** and is w renamed to give all variables an index i. An **indexed substitution** is a substitution with all variables indexed. We now extend the definition of signed input wffs to include an index.

A **signed wff instance (swi)** is a triple $<s,w,i>$ such that $s \in \{p,n\}$ is a sign; w is an input wff; and i is a non negative integer called the index.

Define the **value** $<s,w,i>^*$ of the swi $<s,w,i>$ to be the (indexed) wff w^i if $s=$ 'p' and the (indexed) wff $(\neg w^i)$ if $s=$ 'n'.
This is never actually computed, but will be used to define the semantics for the *PROVEN* relation.

The relation that will be axiomatised in the predicate calculus will be of the form:

$$PROVEN(\alpha,C,\theta)$$

where α is a swi;
C is a set of swi; and
θ is an indexed substitution.

The semantics that we wish to give this relation are

$$facts \vdash [C^* \supset \alpha^*]\theta$$
$$where \quad C^* = \bigwedge_{\gamma \in C} \gamma^*$$

3.1. Predicate Calculus Axiomatisation

For a complete listing of the predicate calculus axiomisation see appendix A.

The rules for the logical connectives transfer straight from the propositional rules with the addition of the same index to each signed wff instance; and adding substitutions to the relations. Where two *PROVEN*s are used to form a third, the substitution of the resultant relation should be the unifying composition[†] of the substitutions of the other two. For example the implication axiom becomes:

if $\{j,k\}=\{1,2\}$ and $PROVEN(<op_0,w,i>,C,\theta)$ and $PROVEN(<op_j,w_j,i>,C,\theta_j)$ then $PROVEN(<op_k,w_k,i>,C,\theta_k)$ where $\theta_k = uc(\theta,\theta_j)$

The Fact rule (2) becomes, simply

$$PROVEN(<p,f,i>,C,\{\})$$

For the case set rules, one must allow for a different instance in the swi and in the case list, together with a substitution that makes them the same. We give the rule here for the substitution appearing in the assuming rule, because it is better for the backward chaining control strategy.

Unassuming Axiom (Rule 5(a))

if $PROVEN(<s,w,i>,C\cup\{<\sim s,w,i>\},\theta)$ *then* $PROVEN(<s,w,i>,C,\theta)$

Assuming Axiom (Rule 5(b))

$PROVEN(<s,w,j>,C\cup\{<s,w,i>\},\theta)$ *where* $\theta=\{v^j/v^i : v \text{ is a variable in } w\}$

The validity of this follows from the fact that θ is the most general unifier of $<s,w,j>$ and $<s,w,i>$, and so under that substitution they are equal. The set of variables in w can be calculated at input time, and no unification or decomposition of the wff w needs to be done at run time.

Equality Substitution Rule (Rule 1)

The only other rule that needs to be lifted to the predicate calculus is the equality substitution rule:

If w and y are unifiable input atomic formulae, with $\phi = mgu(w^0,y^1)$ then if $PROVEN(<s,w,m>,C,\chi)$ then $PROVEN(<s,y,j>,C,\theta)$ where $j \neq m$ and j does not appear in C or χ; and if $\phi^{\{0\to m,1\to j\}}$ is ϕ with all indexes of 0 rewritten to m and

---

† the unifying composition (Sickel[76]) of two substitutions ϕ and θ, written $uc(\phi,\theta)$ is the most general substitution ξ satisfying $\phi o \xi = \xi o \phi = \xi = \theta o \xi = \xi o \theta$. If the unifying composition of two substitutions exists then the subsitutions are said to be consistent. For relevant results see Poole[82].

all indexes of 1 rewritten to j, then $\theta = uc(\chi, \phi^{\{0\to m, 1\to j\}})$

This may be paraphrased as "if w and y can be made the same under some substitution, and an instance of w is proven (with the corresponding substitutions consistent) then an instance of y is proven."

Proof of Correctness:
Let $\phi' = \phi^{\{0\to m, 1\to j\}}$
then $\phi' = mgu(w^m, y^j)$ if $m \neq j$
so $<s,w,m>^*\phi' = <s,y,j>^*\phi'$
so $<s,w,m>^*\phi'\theta = <s,y,j>^*\phi'\theta$
so $<s,x,m>^*\theta = <s,y,j>^*\theta$ -- by definition of "uc"
suppose $PROVEN(<s,w,m>,C,\chi)$
that is $facts \vdash [C \supset <s,w,m>^*]\chi$
so $facts \vdash [C \supset <s,w,m>^*]\theta$ -- by definition of "uc"
so $facts \vdash C\theta \supset <s,w,m>^*\theta$
so $facts \vdash C\theta \supset <s,y,j>^*\theta$ -- by equality above
i.e. $PROVEN(<s,y,j>,C,\theta)$

3.2. Status of the Axiomatisation

The full axiomatisation of the predicate calculus *PROVEN* (as it appears in Appendix A), has been proven both correct and complete in the sense that $PROVEN(w,C,\theta)$ models the relation $facts \vdash [C^* \supset w^*]\theta$, and if $facts \vdash success$ then $PROVEN(<p, w_{success}, 0>, \{\}, \theta)$ (where $w_{success}$ is the implicit input wff representing the atom *success*) follows from the axioms for some θ.

3.3. An Example

Suppose that we have the following facts (with names for the input wffs associated with the principle operator in the fact):

$$\underset{w_1}{P(v)} \ \underset{w_2}{\vee} \ \underset{w_3}{Q(v)}$$

$$\underset{w_4}{P(v)} \ \underset{w_5}{\supset} \ \underset{w_6}{P(f(v))}$$

$$P(\underset{w_7}{f(f(a))}) \ \underset{w_8}{\vee} \ \underset{w_9}{Q(a)} \ \underset{w_{10}}{\supset} \ \underset{w_{11}}{success}$$

Let the object level goal be the atom *success* which is the input wff $w_{success}$. The following deduction can be used:

$PROVEN(<n,w_8,5>,\{<n,w_8,1>\},\{\})$ -- rule 5(a)
$PROVEN(<n,w_9,5>,\{<n,w_8,1>\},\{\})$ -- rule 4(c)
$PROVEN(<n,w_3,4>,\{<n,w_8,1>\},\{a/v_4\})$ -- rule 1
$PROVEN(<p,w_2,4>,\{<n,w_8,1>\},\{\})$ -- rule 2
$PROVEN(<p,w_1,4>,\{<n,w_8,1>\},\{a/v_4\})$ -- rule 4(d)
$PROVEN(<p,w_4,3>,\{<n,w_8,1>\},\{a/v_4,a/v_3\})$ -- rule 1
$PROVEN(<p,w_5,i>,\{<n,w_8,1>\},\{\})$ -- rule 2
$PROVEN(<p,w_6,3>,\{<n,w_8,1>\},\{a/v_4,a/v_3\})$ -- rule 4(d)
$PROVEN(<p,w_4,2>,\{<n,w_8,1>\},\{a/v_4,a/v_3,f(a)/v_2\})$ -- rule 1
$PROVEN(<p,w_6,2>,\{<n,w_8,1>\},\{a/v_4,a/v_3,f(a)/v_2\})$ -- rule 4(d)
$PROVEN(<p,w_7,1>,\{<n,w_8,1>\},\{a/v_4,a/v_3,f(a)/v_2\})$ -- rule 1
$PROVEN(<p,w_8,1>,\{<n,w_8,1>\},\{a/v_4,a/v_3,f(a)/v_2\})$ -- rule 4(a)
$PROVEN(<p,w_8,1>,\{\},\{a/v_4,a/v_3,f(a)/v_2\})$ -- rule 5(b)
$PROVEN(<p,w_{10},1>,\{\},\{\})$ -- rule 2
$PROVEN(<p,w_{11},1>,\{\},\{a/v_4,a/v_3,f(a)/v_2\})$ -- rule 4(d)
$PROVEN(<p,w_{success},0>,\{\},\{a/v_4,a/v_3,f(a)/v_2\})$ -- rule 1

One question that arises from such a proof is "How did you know what to start with?" Backward chaining on the axioms finds the assumption by building up the assumption set as in the propositional case. The use of indexing allows the same variable in two different facts with no confusion as to which variable is meant. Not expanding the third fact into two clauses allows the assumption to be made at an earlier stage.

3.4. The System as a Connection Graph Proof Procedure

An implementation of any system from the axioms can be seen as a connection graph proof procedure. The set of the 'ϕ' for rule 1 forms what is commonly called a connection graph. They all may be evaluated at input time, so that at run time (prove time) no searching for matches or unifications needs to be done. Substitutions will be combined by doing unifying compositions. The substitutions contain only the essential information from the unifications. All possible matches can be found before the search for a solution, relieving the system from many unnecessary searches. This corresponds to the use of connection graphs (eg Bibel[83], Chang and Slagle[79], Sickel[76]) and to compiling the rules of Prospector into an inference network (Duda et al[78]).

3.5. Explanation and Debugging

Any implementation uses the input form of the facts. No input is destroyed, and all deduction steps can be explained directly in terms of input given by the user. This is very important in debugging the knowledge source, as each step can be identified with a particular piece of knowledge. It is also important in explanations, because English explanations can be associated with the facts, and printed out at the appropriate time. Mathematical style proofs may be printed out by using the explanations from a traversal of the solution and-tree, or one can step around a solution tree producing expert system style "*HOW*" and "*WHY*" explanations (eg Emycin - VanMelle[80]). Explanations need not be associated with individual rules, but may be at a higher level, for example, at the level of facts given by the user. Such an implementation has been built (Poole[82]), and tested as an expert system for student enrolment, with English explanations for both mathematical style proofs and Emycin style explanations.

3.6. Adding A Control Component

As in the propositional case any search strategy for horn clauses can be used on the axiomatisation. The backward chaining control structure admits a neat solution to the problem of when to consider cases.

Assigning Unique Indexes

One of the problems that arises in backward chaining is in the choice of "m" in the equality substitution axiom. It can be proven that if there is a global variable $max-index-in-system$, which is incremented by one at each invocation of rule 1, and is the value of m chosen, then j will satisfy the conditions of the rule. This is because each index in C and χ will be greater than or equal to m and $j<m$.

The Search Tree

The search tree for backward chaining on the axioms can be characterised as follows: There is an and/or search tree [nodes corresponding to PROVEN relations; and-arcs formed by rules 4(b) and 4(d); and or-arcs produced by rules 1, 4(a) and 5(a) (the choice of assuming or not assuming); and terminal nodes by rules 2 and 5(a)]. Substitutions are associated with or-arcs, but not with and-arcs. The aim is to find an and-subtree with all substitutions being consistent. At this level this is exactly the structure of the search space of a Prolog or linear resolution system. In fact, given the same input (and considering the Prolog tree as a binary tree), the search tree above has an extra node for each fact used; namely the trivially satisfiable top level of the fact itself. The or-nodes and the corresponding substitutions correspond exactly. Our system will need to do a case analysis at each node, but this seems to be a small price to pay for the use of many different operators. The only other thing that our system (like a linear resolution system) must do over what the Prolog system does is to maintain the case set (but if it only wants the power of a pure Prolog it does not even need to do this).

The Case Set

Given such a definition of the control, the case set does not need to be explicitly stored, but is (the negation of) the set of ancestors of a node. The unassuming axiom thus becomes implicit in building the tree. The use of the assuming axiom corresponds to tautological circuits (Bibel[82]), to merge condition (Andrews[76]), Tautology Loop (Sickel[76]), and to reduction of reducible ordered clause in Linear Resolution (Chang and Lee[73]).

Modularising the Implementation

The abstract definition of the search tree points to one possible way to implement the system, with the logic part and the control part implemented as separate modules. The and/or tree becomes the communications interface; the control part searching the and/or tree and the logic part building the tree as needed. We can implement our system, Prolog or linear resolution by changing the logic component, independent of the control structure.

Use of the Substitutions

There are a number of ways that substitutions can be evaluated when backchaining on the axioms.

The first is to do all unifying compositions at the end. This can be done by treating *uc* as an uninterpreted function during the search process. A solution can be found with the resulting substitution being an expression involving the function *uc*, as well as reindexed input substitutions. This expression can then be evaluated, and if it is inconsistent another solution can be looked for. This corresponds to having a planning phase and a verification phase, and has been advocated by a number of people (Chang and Slagle[79], Klahr[78]).

The second is to evaluate the unifying composition when all of its arguments have been evaluated. This corresponds to *uc* being a predicate on the antecedent of Prolog type systems.

The third is to use the associativity, commutativity and idempotency of unifying composition (Poole[82]) to realise that the ordering in which the substitutions are combined is irrelevant. A global substitution can be maintained; new substitutions only being added if they are consistent with every substitution in the system. This allows any inconsistencies to be located as soon as possible. Any substitution must be consistent with any other substitution in a solution and-tree, and in particular must be consistent with all ancestor substitutions. Such considerations lead directly to the notion of non-chronological backtracking. (See Poole[82] for the correctness proof of such a control component.)

4. Bidirectional Search

A goal directed search strategy at the meta level (as outlined above) does not necessarily impose a goal directed search strategy at the object level. If the top level goal at the meta level is the negation of a fact, subject to instances of the negation of *success*, then a forward chaining proof develops.

A bidirectional searching mechanism can be built be changing the logic component. To extend the logic we define a new relation

$$CANCEL(\alpha_1,\alpha_2,C,\theta)$$

where α_1 and α_2 are swi, C is a case set, and θ is an indexed substitution. It is named *CANCEL* after the relation in Nilsson[79]. It is to represent "if α_1 is true then α_2 is true". The "α_1" is the "facts" implying "α_2" the "goals". More formally the semantics will mean:

$$facts \vdash [C^* \wedge \alpha_1{}^* \supset \alpha_2{}^*]\theta$$

Each of the rules for *PROVEN* can be extended to rules for *CANCEL* by making the first argument of the *CANCEL* free. There is also the equivalence:

$$CANCEL(<s_1,w_1,i_1>,<s_2,w_2,i_2>,C,\theta) \equiv CANCEL(<\sim s_2,w_2,i_2>,<\sim s_1,w_1,i_1>,C,\theta)$$

This equivalence can be used in two ways: It can be applied to the rules to produce another set of rules for forward chaining as well as backward chaining; alternatively it can be used as a rule itself, in which case it acts as a change of focus for the problem solver. That is alternating between "considering the goals" to "considering the facts". We can also define a rule to let us join the facts to the goals, and this rule is:

If w and y are unifiable input atomic formulae, with $\phi = mgu(w^0,y^1)$ then

$$CANCEL(<s,w,i>,<s,y,j>,A,\phi^{\{0\to i,1\to j\}})$$

Such a system with this *CANCEL* relation forms a complete extension of Nilsson[79]'s production system for automatic deduction; with all of the advantages of PROVEN in a system with bidirectional search (it is however arguably more difficult to control).

5. A Comparison With Other Systems

5.1. Other Meta-level Definitions

The main difference between this approach and other more traditional approaches to the definition of inference rules (eg. Manna[74], Bowen[82]) is in the direct translation into efficient implementation. Our axiomatisation shows the information necessary to define new operators within such an implementation.

The most similar machine oriented logical definition of a deduction system is that of Bowen and Kowalski[82]. Our system can be seen as an extension of their definition of a horn clause problem solver to the complete first order predicate calculus. Our definition differs in that it is meant to be the formal definition of a deduction system, and does not necessarily appear within the problem solver itself.

5.2. Prolog

If the input to our system is restricted to Horn Clauses then the and/or tree produced is identical to that produced by Prolog. If a left to right search strategy with chronological backtracking is used then Prolog results. This paper shows how Prolog implementations can be expanded to cover the complete first order predicate calculus, with the resulting cost being the need to search for oppositely signed ancestors in the search tree. Our system is not constrained to use such search strategies though.

5.3. Connection Graph Proof Procedures

Despite having a completely different derivation and justification, the resulting system is remarkably similar to the modern "connection graph" theorem proving systems (eg Bibel[81,83], Andrews[76,81], Sickel[76]), and shares their advantages of efficiency due to elimination of redundancy.

Building a consistent and-tree can be seen as a systematic way of checking all possible paths in a connection matrix of Andrews or Bibel. The use of rule 1 corresponds to a mating or a connection. The and-offspring correspond to the generation of other possible paths not through that mated literal. This formalises the informal description of systematising the p-acceptable criteria in Andrews[81] (pg 204).

Sickel[76]'s search technique for Clause Interconnectivity Graphs can be seen as an implementation of our system restricted to clauses. Her "walk" corresponds to a branch in our and-tree; her "markers" to the leaves in the current and-tree; and her "merge loop" corresponds to the assuming rule.

Our system has a number advantages over these other systems. We do not need clause form, or negation normal form or matrix representation, but can work on the input wffs directly. In particular we need not expand out equivalences. The Search strategy is less constrained. We don't need to do chronological backtracking, so for example, we can combine the efficiency of connection methods with the efficiency of non-chronological backtracking. The quantifier duplication problem (Andrews[81] p 207) and the index increasing technique that poses a problem (Bibel[82] pg 210) has been neatly solved. Most importantly our system is simpler and easier to comprehend. This can be seen at the global level where the proof is subgoaling, and at the local level as each step is explainable as a simple local logical operation in terms of the input given by the user.

Our description is at a lower level than the description of the other systems, so a detailed description will be more detailed. The search trees will be very similar with our and-tree being a binary tree, with a CANCEL at each node. The operation at each node is a trivial recognition of its type. The and-nodes will be the same as the other systems of their trees are considered as binary trees. Note however that there are extra nodes for the (trivially solved) fact, and for explicit negations (not allowed in other systems). The or nodes due to connections (rule 1) will be identical.

6. Conclusion

This paper discusses the design of whole families of problem solvers with a clear separation of logic and control.

- The logic part has been proven correct and complete independent of any control imposed.
- The logic is defined for the full first order predicate calculus, and may be augmented to include other logical or non logical connectives. (For example, one does not need to be very imaginative to invent a rule for exclusive-or, or a one way implication of the form $x \rightarrow y$ which can be used to deduce y from x, but not $\neg x$ from $\neg y$.)
- The logic has been defined in such a way that implementations may be directly derived. It also shows what information is sufficient to define an operator within such a deduction system.
- The problem solvers produced do not change the input given by the user, and the steps are simple uses of the logical connectives, thus enhancing explanation and debugging. The explanation and debugging facilities of expert systems can be incorporated into such systems.
- The formalism allows the comparison and combination of many previous systems (eg Linear Resolution, Prolog, Sickel[76], Andrews[76,81], Bibel[82,83]).
- The problem solver can be expanded to include bidirectional search.
- The logic component of the problem solver can be extended to more powerful logics, for example it has been expanded to include defaults (Poole[84]), as well as a proposal to extend it to cover a logic of actions (Poole[82]).

- The logic and control can be separated even at the implementation level, where they can be hived off to different modules. Such a system has been designed and proven correct (Poole[82]).
- It forms a powerful vehicle for the systematic study of domain specific control strategies. Using input form with a clear separation of logic and control has the potential to allow the user to associate control knowlege with the object level facts, and for the system to use that knowledge effectively. This system is currently being implemented to investigate control primitives to express effective heuristics and strategies.

Appendix A - Complete Axiomatisation of Predicate Calculus PROVEN

1. Equality Substitution Rule

If w and y are unifiable input atomic formulae, with $\phi = mgu(w^0, y^1)$

$$\text{if } PROVEN(<s,w,m>,C,\chi) \textit{ then } PROVEN(<s,y,j>,C,\theta)$$

where $j \neq m$ and j does not appear in C or χ; and $\theta = uc(\chi, \phi^{\{0 \to m, 1 \to j\}})$

2. Fact Rule

If f is a fact then

$$PROVEN(<p,f,i>,C,\{\})$$

3. Negation Rules

If w is of the form $(\neg z)$ then

3(a) Upward Negation Rule

$$\text{if } PROVEN(<s,z,i>,C,\theta) \textit{ then } PROVEN(<\sim s,w,i>,C,\theta)$$

3(b) Downward Negation Rule

$$\text{if } PROVEN(<s,w,i>,C,\theta) \textit{ then } PROVEN(<\sim s,z,i>,C,\theta)$$

4. Binary Logical Connectives

If w is of the form $(w_1 \ op \ w_2)$ then

4(a) Upward Disjunction Rule

$$\text{for } j \in \{1,2\} \text{ if } PROVEN(<op_j,w_j,i>,C,\theta) \textit{ then } PROVEN(<op_0,w,i>,C,\theta)$$

4(b) Upward Conjunction Rule

$$\text{if } PROVEN(<\sim op_1,w_1,i>,C,\theta_1) \textit{ and } PROVEN(<\sim op_2,w_2,i>,C,\theta_2)$$

$$\textit{then } PROVEN(<\sim op_0,w,i>,C,\theta) \ \textit{ where } \theta = uc(\theta_1,\theta_2)$$

4(c) Downward Disjunction Rule

for $j \in \{1,2\}$, if $PROVEN(<\sim op_0,w,i>,C,\theta)$ *then* $PROVEN(<\sim op_j,w_j,i>,C,\theta)$

4(d) Implication Rule

for $\{j,k\}=\{1,2\}$, if $PROVEN(<op_0,w,i>,C,\theta)$ *and* $PROVEN(<\sim op_j,w_j,i>,C,\theta_j)$

then $PROVEN(<op_k,w_k,i>,C,\theta_k)$ *where* $\theta_k = uc(\theta,\theta_j)$

4(e) Equivalence Rule

If w_0 is of the form $w_1 \equiv w_2$ and $\{j,k,l\}=\{0,1,2\}$ and $PROVEN(<s_j,w_j,i>,C,\theta_j)$ and $PROVEN(<s_k,w_k,i>,C,\theta_k)$ then $PROVEN(<s_l,w_l,i>,C,\theta_l)$ where $s_l = (if\ s_j = s_k\ then\ 'p'\ else\ 'n')$

5. Case Set Rules

5(a) The Assuming Rule

$PROVEN(<s,w,j>,C \cup \{<s,w,i>\},\theta)$ *where* $\theta = \{v^j/v^i : v \text{ is a variable in } w\}$

5(b) The Unassuming Rule

if $PROVEN(<s,w,i>,C \cup \{<\sim s,w,i>\},\theta)$ *then* $PROVEN(<s,w,i>,C,\theta)$

Appendix B - Theorems Proven About PROVEN

The following theorems have been proven (Poole[82]) about the PROVEN relation. For details see Poole[82].

Theorem 1: If

$PROVEN(<s_1,w_1>,\{<s_2,w_2>\} \cup A)$

can be deduced then

$PROVEN(<\sim s_2,w_2>,\{<\sim s_1,w_1>\} \cup A)$

can be deduced using a deduction with no more applications of rules other than 5(a).

This is proven by a constructive proof by induction on the length of the deduction. The induction step is proven by considering each possible last step in the deduction of $PROVEN(<s_1,w_1>,\{<s_2,w_2>\} \cup A)$. The base step must be an instance of rule 5(a).

Theorem 2: If

$PROVEN(<s,w>,\{<\sim s,w>\} \cup A)$

is set up as a subgoal in a backward chaining proof, then it can be removed without affecting completeness. Moreover, removing it will not remove the shortest proof (shortest in the number of applications of rules other than 5(a).

This is proven by using Theorem 1 to construct a shorter proof that does not use the above form from one that does.

Theorem 3: Completeness of the ground case. If F is a set of satisfiable facts in the propositional calculus, and $F \vdash success$ then

$$PROVEN(<p, w_{success}>, \{\})$$

can be deduced.

This is proven by induction on the number of atomic symbols in F. The ground case of the induction is proven by induction on the height of the parse tree of the formula.

Theorem 4: Completeness for the predicate calculus. If F is a set of facts in the predicate calculus, such that F is satisfiable, and $F \vdash w_{success}$ then

$$PROVEN(<p, w_{success}>, \{\}, \theta)$$

can be deduced for some θ.

Proof of this is by lifting a proof from the ground case.

Acknowledgements

The major part of this work was carried out when the author was at the Australian National University. Special thanks to Robin Stanton of the Australian National University who supervised my thesis. Thanks also to Hugh MacKenzie and Peter Creasy for many discussions and to Randy Goebel, Marlene Colbourn and Romas Aleliunas for valuable comments on an earlier draft of this paper. This research was supported by absolutely no contracts from any defence department.

References

Andrews,P.B. [76], "Refutations by Matings", *IEEE Transactions on Computers,* C-25, pp 801-807.

Andrews,P.B. [81], "Theorem Proving via General Matings", *Journal of the Association for Computing Machinery,* Vol 28, No 2, April 1981, pp 193-214.

Bibel,W. [82]. *Automated Theorem Proving,* Vieweg: Braunschweig, 1982.

Bibel,W. [83] "Matings in Matrices", *Comm A.C.M.* Vol 26, No 11, Nov 1983.

Bowen,K.A. [82], "Programming with full first-order logic", in Hayes,J.E. and Michie,D. (Eds) *Machine Intelligence 10,* Ellis Horwood, Chichester, 1982.

Bowen,K.A. and Kowalski,R.A.[82], "Amalgamating Language and Metalanguage in Logic Programming", in Clark,K.L. and Tarnlund,S.-A. (Eds) *Logic Programming,* Academic Press, London, pp 153-172.

Chang,C. and Lee,R.C. [73], *Symbolic Logic and Mechanical Theorem Proving,* Academic Press, New York, 1973.

Chang,C. and Slagle,J. [79], "Using Rewriting Rules for Connection Graphs to Prove Theorems", *Artificial Intelligence* Vol 12, No 2, pp 159-178.

Davis,R. [80] "Meta Rules: reasoning about Control", *Artificial Intelligence,* vol 15 (1980), pp 179-222.

Duda,R.O.; Hart,P.E.; Nilsson,N.J. and Sutherland,G.L. [78], "Semantic network representations in rule based inference systems", in Waterman,D.A. and Hayes-Roth (Eds) *Pattern Directed Inference Systems,* Academic Press, NY 1978.

Georgeff,M.P. [82], "Procedural Control in Production Systems", *Artificial Intelligence,* vol 18 (1982), pp 175-201.

Klahr,P. [78] "Planning Tequniques for Rule Selection in deductive question-answering" in Waterman,D.A. and Hayes-Roth,F. (Eds) *Pattern Directed Inference Systems* Academic Press, NY 1978, pp 223-239.

Kowalski,R. [79], "Algorithm = Logic + Control", *Comm.A.C.M.* Vol 22, No 7, July 1979, pp 424-436.

McCarthy,J. and Hayes,P. [69], "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in Meltzer,B. and Michie,D. (eds) *Machine Intelligence 4,* pp 463-502.

Manna,Z. [74], *Mathematical Theory of Computation,* McGraw-Hill, NY 1974.

Nilsson,N.J. [79], "A Production System for Automatic Deduction", in Hayes,J.E., Michie,D. and Mukulich,L.I. (Eds.) *Machine Intelligence 9: Machine Expertise and the Human Interface,* Ellis Horwood, Chichester, 1979.

Poole,D.L. [82], *The Theory of CES: A Complete Expert System,* Ph.D. Dissertation, Department of Computer Science, Australian National University, October 1982.

Poole,D.L. [84], *A Computational Logic of Default Reasoning,* Reseach Report, Department of Computer Science, University of Waterloo.

Sickel,S. [76], "A search Technique for Clause Interconnectivity Graphs", *I.E.E.E. Transactions on Computers,* Vol C-25, No 8, August 1976, pp 823-835.

van Melle,W. [80] *A Domain Independent System that Aids in Constructing Knowledge Based Consultation Programs,* Stanford University Computer Science Report 80-820, 1980.