WORKING MATERIAL

FOR THE LECTURES OF

J. ALAN ROBINSON

RESOLUTION

INTERNATIONAL SUMMER SCHOOL

ON

LOGIC, ALGEBRA AND COMPUTATION

MARKTOBERDORF, GERMANY, JULY 25 - AUGUST 6, 1989

THIS SUMMER SCHOOL IS ORGANIZED UNDER THE AUSPICES OF THE TECHNISCHE UNIVERSITÄT MÜNCHEN AND IS SPONSORED BY THE NATO SCIENCE COMMITTEE AS PART OF THE 1989 ADVANCED STUDY INSTITUTES PROGRAMME. PARTIAL SUPPORT FOR THE CONFERENCE WAS PROVIDED BY THE EUROPEAN RESEARCH OFFICE AND THE NATIONAL SCIENCE FOUNDATION AND BY VARIOUS INDUSTRIAL COMPANIES.

Resolution

J. A. Robinson

Syracuse University

May 1989

Introduction. These notes form a brief but reasonably complete account of the ideas underlying resolution. We try to give equal emphasis not only to the logical principles involved but also to the computational issues which arise. We therefore study in some detail the most useful symbolic algorithms (in particular, since it is of greatest importance, the unification algorithm) and treat logical formulas explicitly as carefully engineered data structures. One of our aims is to explain resolution in general in such a way that the important special case of Horn clause resolution can be properly understood within the broader setting. Horn clause resolution is the theoretical framework for the kind of logic programming which is done by users of PROLOG. Indeed many of the ideas we shall discuss are concretely realized, although not always in the purest form, in various versions of PROLOG. We shall not be concerned specifically with PROLOG, however, since the surface details often vary considerably from version to version and are often complex enough to hide the relatively simple conceptual system which lies just below the surface.

Formal symbolic computation. Expressions as data structures. Logic programming is a technique for specifying formal symbolic computations, that is, computations with formal symbolic expressions as data objects. Logicians have long dealt with expressions in this way, and in computer science one must adopt the same approach in dealing formally with programming languages. In formal logic one deals with expressions as formal structured objects, to be disassembled and constructed according to mathematically precise procedures. Thus in particular in the automatic theorem proving problem for the predicate calculus one studies algorithms which, when given (as input) a sentence S will construct (as output) a proof P of S (provided that S is logically true). Such an algorithm may sometimes (but not always, because of the impossibility of a decision procedure for logical truth in the predicate calculus) be able to indicate correctly that S is *not* logically true if that is indeed the case. Both S and P have to be treated as data structures.

In most theorem proving or logic programming applications the formal expressions will have an informal (perhaps even a formal) semantics - a logic programmer seeking to generate a set of "answer expressions" or a mathematician seeking to demonstrate a theorem will usually mean something by them - but in formal theorem proving and in logic programs the object expressions are treated purely formally, that is to say, as structured, manipulable syntactic objects of whose meaning, if any, no "official" notice is taken at all. Only their (abstract) *form* is used as a basis for both their analysis and synthesis. This point of view will of course already be especially familiar to those who have used LISP or (of course) PROLOG.

Formal expressions. Atoms = constants + variables. Indeed in dealing formally with expressions we find it very convenient to use the universal but simple and convenient ontology of LISP. Accordingly we shall take expressions to be objects generated from two countably infinite, disjoint sets, the set CONST of constants and the set VARS of variables, by the single binary operation, dot. The constants and variables are also known as atoms . Thus we have a countably infinite set ATOMS which is just CONST ∪ VARS. Constants and variables are usually concretely represented by finite strings over some suitable alphabet of characters. It does not then much matter what these characters are, but to avoid confusion the various bracketing characters < > { } ( ) [ ] should not be among them, nor the space, nor the the comma, nor the character reserved for the dot operation: • . In what follows we shall write variables as strings of lower case letters, possibly subscripted, as for example: x, reverse, y_2. All other strings are constants (this includes numerals, arithmetic operators, strings containing upper case letters, and so on). So: atoms are expressions. But there is another kind of expression, the so-called cons.

Conses. If A and B are expressions, then so (to use the terminology of LISP) is the **cons** P whose **car** is A and whose **cdr** is B, and we write:

P = [• A B]	("P is the cons of A and B")
aP = A	("the car of P is A")
dP = B	("the cdr of P is B").

Remarks: "cons" rhymes with "once" and is singular; "cdr" is pronounced "kidder" by some and "kudder" by others; the point of these improbable coinages is to preserve such words as "pair", "head", tail" for other purposes, and to mark and enforce the distinction between the general syntax of expressions-as-such and the particular syntax (to which we shall soon come) of the predicate calculus. The original LISP convention was to write the dot infixed, [A • B] rather than [• A B]. In general [• A B] is the same expression as [• B A] only when A and B are the same expression. Nor is the expression [• A[• B C]] ever the same as the expression [•[• A B]C].

Lists. The general syntax also contains the notion of a **list**. Certain constants are given special roles, most notably the constant NIL, whose role as the empty list is introduced in the following general definition, in which we define several notions simultaneously:

- the constant NIL is a **list**; moreover it is the *(only)* ***empty list***, and it has **length** 0;
- for any list L of length n ≥ 0 and any expression A, the cons [• A L] is a list of length n+1; the 1st **element** of [• A L] is A and the (j+1)st element of [• A L] is the jth element of L, for j=1, . . ., n.

So NIL is the only list which is an atom All lists of nonzero length are conses. Of course, not all conses are lists: only (but all) those conses whose **cdrs** are lists are themselves lists.

List notation. Although the dot notation is in principle completely adequate for all purposes, we shall mostly be dealing with lists and will therefore use LISP's more intuitive and flexible list notation. This is based on the further convention that allows nested conses of the form

$$[\cdot\, A_1 [\cdot\, A_2 \ \ldots [\cdot\, A_n \text{ NIL}] \ldots]],$$

namely nonempty lists, to be written without the dots or interior parentheses, thus:

$$[A_1\ A_2 \ldots A_n].$$

The list notation also allows NIL to be written alternatively and intuitively as: [ ].

So to sum up the general syntax of expressions we have:

expressions = conses + atoms
= conses + constants + variables.

Substitutions. We shall have much to do with certain mappings of expressions to expressions, called substitutions. Intuitively, the image of an expression E under a substitution θ is the expression which we can construct by replacing each occurrence of each of a certain set of variables within E by another expression (possibly by another variable). Each occurrence of the same variable is replaced by an occurrence of the same expression (called the value of that variable under θ). More precisely, the behavior of a substitution θ is completely defined, because of the "evaluation" rule given below, by its behavior on the variables.

Formal definition of application of substitutions. For each expression E, the expression $E\theta$ onto which θ maps E is called the instance of E by θ The following "instantiation" rule then says, intuitively, that the expression $E\theta$ is the result of simultaneously replacing each variable in E by the expression which is the instance of that variable by θ. The constants in E are left alone. The rule is very simple and has three cases:

(1) $E\theta$ = E when E is a constant ;
(2) = $[\cdot\ A\theta\ B\theta]$ when E is the cons [• A B]
(3) = the instance of E under θ when E is a variable.

Bindings. Descriptions of substitutions. In fact, many variables may be mapped onto themselves by θ, and so will be treated by part (3) of the rule exactly as part (1) treats constants. This means that the instantiation rule allows us to construct $E\theta$, for any expression E, if we know the instance $X\theta$ of each variable X whose instance by θ is different from the variable itself. We say these variables are bound by θ, and that the couples $< X\ X\theta >$ are the bindings of θ. Finally, in order to have a smooth, flexible notation for describing substitutions, we say that any set of couples of the form $< X\ X\theta >$ (where X is a variable) which contains all the bindings of θ, is a description of θ. Any couple in a description of θ which is *not* a binding of θ must have the trivial form $< X\ X >$. If we discard these we get the unique minimal description of θ.

It is the usual convention, which we have been following above, to use lower case Greek letters to denote substitutions, and to indicate their application to expressions by juxtaposition on the right.

We shall frequently indulge in a harmless abuse of notation in which a substitution is identified with a description of it. Thus, we shall write, for example,

$$\theta = \{< y\ [A\ x]>,\ \ < x\ 5>,\ \ < z\ [B\ y]>\}$$

to mean that θ is the substitution described by the right hand side. Then for example we can easily verify that

$$[A\ [B\ x\ y]\ [C\ x\ y\ z\ w]]\theta = [A\ [B\ 5\ [A\ x]]\ [C\ 5\ [A\ x]\ [B\ y]\ w]].$$

We shall also find it convenient to extend the instantiation notation and write $S\theta$, where S is a set of expressions, to denote the set of all expressions $X\theta$, where X is in S. It is very important then to note that the set $S\theta$ may have fewer members than the set S, since it is possible that $X\theta$ and $Y\theta$ should be the same expression even though $X \neq Y$. Indeed, this is what is meant by saying that X and Y are unified by θ.

Composition of substitutions. The product $\theta\lambda$ of two substitutions θ and λ is simply their composition as mappings, that is, $\theta\lambda$ is the substitution which maps each expression E onto the expression $(E\theta)\lambda$. Thus the product operation is associative, and has an identity, namely the substitution which maps every expression onto itself. The usual notational convention, which we shall follow, is to denote the identity substitution by the lower case Greek letter ε. Note that the empty set $\{\}$ is then a description of ε. Not all substitutions are bijections, of course, but when θ *is* a bijection we shall write θ^{-1} for its inverse. We then have $\theta\theta^{-1} = \theta^{-1}\theta = \varepsilon$. A bijective substitution is called a change of variables.

Descriptions of products. Since in computations substitutions are represented by their descriptions, we shall often need to construct a description of a product

$\theta\lambda$, given descriptions of θ and λ. We then use the following easily verified fact:

- if V and W are the sets of variables bound by θ and λ respectively, then the set of couples

  $$\{ \langle X\ (X\theta)\lambda \rangle \mid X \text{ in } (V \cup W) \}$$

  is a description of $\theta\lambda$.

In order to write a minimal description of $\theta\lambda$ we first construct this set and then drop from it the trivial couples (if any) of the form: <V V>. So, for example, the product of the substitution

$$\theta = \{ \langle x\ [A\ x\ y] \rangle,\ \langle v\ w \rangle \}$$

with the substitution

$$\lambda = \{ \langle x\ [C\ y] \rangle, \langle y\ [B\ z\ x] \rangle,\ \langle w\ v \rangle \}$$

is (dropping the trivial couple $\langle v\ v \rangle$) the substitution

$$\theta\lambda = \{ \langle x\ [A\ [C\ y]\ [B\ z\ x]] \rangle,\ \langle y\ [B\ z\ x] \rangle \}$$

and their product in the reverse order is the substitution

$$\lambda\theta = \{ \langle x\ [C\ y] \rangle,\ \langle y\ [B\ z\ [A\ x\ y]] \rangle \}.$$

Of course, as this example illustrates, substitutions do not in general commute.

Unification. The central concept of the theory of instantation is that of unification. Consider the following unification problem:

- Given any finite set S of expressions find (if one exists) a substitution θ which maps every expression in S onto the same expression (or what is the same, a substitution θ such that $S\theta$ is a singleton) or show (if no such substitution exists) that no such substution exists.

A positive solution θ of this problem is said to unify, or to be a unifier of, the set S and the set S is said to be unifiable. The expression in the singleton $S\theta$ is thus common instance of all the expressions in S.

A unifiable set of expressions may have many, even infinitely many, different unifiers. However, if in the above problem we ask that the substitution θ be not just a unifier of S, but a most general unifier of S, then the solution is essentially unique if there is one. Indeed, if θ is any most general unifier of S, then the set of *all* the most general unifiers of S is just

$$\{ \theta\mu \mid \mu \text{ is a change of variables} \}.$$

Most general unifiers. A substitution σ is a most general unifier of a set S of expressions if it is a unifier of S with the additional property that *any* unifier θ of S satisfies an equation $\theta = \sigma\lambda$ for some substitution λ. Intuitively: the unifier θ is simply an instance (or special case) of a most general unifier of S. Of all the ideas needed for an understanding of resolution and logic programming this idea of most general unification is undoubtedly the most important.

Example of unification. The set { A, B } whose two members are the expressions

A = [[P x y u] [P y z v] [P x v w] [P [K t] t [K t]]]

and

B = [[P [G r s] r s] [P a [H a b] b] [P x y u] [P u z w]]

is unified by the substitution

σ = { <x [G r [K [H r r]]] > <y r> <z [H r r] >
<u [K [H r r]] > <v r> <w [K [H r r]] >
<a r> <b r> <s [K [H r r]] > <t [H r r] > },

as may be readily verified by applying σ to A and B and comparing the results. Indeed σ maps both A and B onto the expression:

[[P [G r [K [H r r]]] r [K [H r r]]]
[P r [H r r] r]
[P [G r [K [H r r]]] r [K [H r r]]]
[P [K [H r r]] [H r r] [K [H r r]]]].

Now, as it happens, σ is also a most general unifier of {A B}: any other unifier θ of {A B} *is a product σ{< r E >} of σ with a substitution* which maps the variable r onto any expression E whatsoever. In particular, if E is a variable, then θ will also be a most general unifier of A and B.

Examples of nonunifiable sets. On the other hand, for example, the set

{ [P x y u] [Q a b c] }

is <u>not</u> *unifiable. It is not difficult to see why this is so:* any unifier would have to unify the two constants P and Q, which is impossible. Again, the set

{ [F x] x }

is not unifiable: any unifier would have to map the variable x into an expression which contained itself as a proper subexpression, which is impossible. These two reasons for nonunifiability turn out to be the only two that there are.

A simple unification algorithm. We next give a simple <u>binary unification algorithm</u> which solves all "binary" unification problems by finding a most general unifier for any set of *two* expressions (if the set is unifiable), or detecting its nonunifiability (if it is not unifiable). Bigger finite sets can then be handled iteratively, by using the (easily verified) fact that

- θλ is a most general unifier of a set S of $n \geq 2$ expressions among which are A and B, if θ is a most general unifier of the set {A, B}, and λ *is a most general unifier of the set* Sθ.

Note that if {A, B} is unifiable then Sθ will have at most (n - 1) elements, since Aθ = Bθ, and thus the iteration will stop after at most (n - 1) steps.

Differences between expressions. To help in formulating the binary unification algorithm in a simple and intelligible (but inefficient) way we shall use the notion of the difference $\Delta(X,Y)$ between any two expressions X and Y.

Intuitively, the difference between X and Y is the set of all unordered pairs of expressions which occur opposite each other at corresponding positions in X and Y where X and Y are not the same. We have the following recursive characterization:

$\Delta(X,Y) = \{\}$ if X and Y are the same expression,
$= \Delta(\mathbf{a}X, \mathbf{a}Y) \cup \Delta(\mathbf{d}X, \mathbf{d}Y)$ if A and B are both conses,
$= \{\{X,Y\}\}$ otherwise.

Negotiability. Reductions. Such a difference is said to be negotiable if

(1) it is nonempty
(2) each pair in it has the property that at least one of its members is a variable and neither member occurs in the other.

Thus, because of this condition, for every member {U, V} of a negotiable difference, at least one (and possible both) of {<U V>} or {<V U>} must be a substitution. These substitutions are called reductions of the difference.

Examples of differences and their reductions. The difference

$\Delta([P\ x\ y\ z], [Q\ a\ b\ c])$

is the set

$\{\{P, Q\}\ \{x, a\}\ \{y, b\}\ \{z, c\}\}$.

This is not negotiable, because it contains the pair {P,Q} and neither of P, Q is a variable. Nor is the difference

$\Delta([F\ x], x) = \{\{[F\ x], x\}\}$

negotiable, because x occurs in [F x]. On the other hand, the difference

$\Delta([P\ x], [y\ Q]) = \{\{P, y\}\ \{x, Q\}\}$

is negotiable, and has two reductions

$\mu = \{<y\ R>\}$ and $\nu = \{<x\ Q>\}$.

Unifiers eliminate differences. The intuitive content of the following proposition is then that the difference between distinct but unifiable expressions is always eliminable:

- **Negotiability Lemma.** If A and B are distinct expressions, and θ unifies {A, B}, then $\Delta(A, B)$ is negotiable, and θ unifies each member of $\Delta(A, B)$.

We now state the (slow!)

- **Binary Unification Algorithm**, for two expressions A, B as input:
  1 $\sigma \leftarrow \varepsilon$;
  2 **while** $\Delta(A\sigma, B\sigma)$ is negotiable **do** $\sigma \leftarrow \sigma\mu$
  where μ is any reduction of $\Delta(A\sigma, B\sigma)$;
  3 **return** (**if** $\Delta(A\sigma, B\sigma)$ is empty **then** σ **else** "FAIL").

We are assured that this algorithm deserves its name by the

> **Binary Unification Theorem.** Let A and B be any two expressions. Then {A B} is unifiable if and only if the above Binary Unification Algorithm terminates, when applied to A and B as input, without returning "FAIL". The σ then returned is, moreover, a most general unifier of {A, B}.

The idea of the proof: termination. At each repetition of the loop in step 2 the number of distinct variables in the expressions $A\sigma$, $B\sigma$ decreases by 1 (since each successive reduction $\{<U\ V>\}$ eliminates the variable U in view of the fact that U does not occur in V). Hence step 2 must terminate after no more repetitions than there are distinct variables in the input expressions A and B.

The idea of the proof: correctness. If, at step 3, $\Delta(A\sigma, B\sigma)$ is empty, then (obviously) the σ returned as output is a unifier of A and B, but it must be shown that it is a most general unifier of A and B. Well: this follows from the fact that for every unifier θ of {A,B} the equation

$$\theta = \sigma\theta$$

is an invariant of the computation. It clearly (indeed, trivially) holds after step 1. We can then see that it is preserved throughout step 2 if we note that for any reduction μ of $\Delta(A\sigma, B\sigma)$ the equation $\theta = \mu\theta$ holds, allowing us to calculate:

$$\theta = \sigma\theta = \sigma(\mu\theta) = (\sigma\mu)\theta.$$

Since σ is replaced by $\sigma\mu$ at each repetition the equation $\theta = \sigma\theta$ is an invariant. Thus for every unifier θ we have $\theta = \sigma\lambda$, with $\lambda = \theta$.

Faster binary unification algorithms. The slow binary unification algorithm given above has the pedagogical and theoretical advantage of being concise and intuitive. However, it is indeed slow. It is very inefficient in both time and space. For example, to unify with this algorithm the two expressions

$$\begin{matrix} [& [F\ x_0\ x_0] & [F\ x_1\ x_1] & [F\ x_2\ x_2] & \ldots & [F\ x_{n-1}\ x_{n-1}]\ ] \\ [& x_1 & x_2 & x_3 & \ldots & x_n\ ] \end{matrix}$$

requires time and space proportional to 2^n. It is easy to see that the most general unifier is

$\{\langle x_1\ [F\ x_0\ x_0]\rangle$
$\langle x_2\ [F\ [F\ x_0\ x_0]\ [F\ x_0\ x_0]\ ]\rangle$
$\langle x_3\ [F\ [F\ [F\ x_0\ x_0]\ [F\ x_0\ x_0]\ ]\ [F\ [F\ x_0\ x_0]\ [F\ x_0\ x_0]\ ]$
:
$\langle x_n$ (an expression containing "F" 2n+1 times and "x_0" 2^n times) $\rangle$ $\}$.

Fortunately it is possible to exploit other aspects of unification in such a way that much faster unification algorithms can be devised. The general idea behind all of them is based on the equivalence relations which are induced on sets of expressions by substitutions.

Occurrence graphs. If S is any set of expressions, the extent of S is the smallest set of expressions which includes S and contains A and B whenever it contains [• A B]. The occurrence graph of S is then the directed graph whose nodes are the singleton sets {X} where X is an expression in the extent of S, with an arc labelled **a** (respectively **d**) leaving the node {[• A B]} and impinging on the node {A} (respectively {B}). If X is a constant or a variable the the node {X} has no arc leaving it. Clearly, the occurrence graph of S is acyclic, but is not necessarily a tree because of the possibility of "sharing".

For each substitution θ we define the equivalence relation $\equiv_\theta$ by:

$X \equiv_\theta Y$ if and only if $X\theta = Y\theta$.

Now if in the occurrence graph G of Sθ we replace each of its nodes {Xθ} by the $\equiv_\theta$-equivalence class in which X lies, we get a graph H which is isomorphic to G. Since G is acyclic, so is H. Let us call H the occurrence graph of $S/\equiv_\theta$. Intuitively, $H\theta = G$. The equivalence relation $\equiv_\theta$ therefore satisfies the two properties:

no cycles the occurrence graph of $S/\equiv_\theta$ is acyclic
structure **for all** distinct X, Y in S:
if $X \equiv_\theta Y$ **then** **either** {X, Y} contains a variable
or X, Y are both dotted pairs
and $aX \equiv_\theta aY$ and $dX \equiv_\theta dY$.

The converse also holds: any equivalence relation ≡ on the extent of S which satisfies the two conditions *no cycles* and *structure* is inducible on the extent of S by a substitution. Indeed, a most general substitution σ which induces ≡ on the extent of S can be immediately constructed from ≡ .

Set union algorithm. The key idea of the various known fast binary unification algorithms is to seek to construct, given distinct expressions A and B, an equivalence relation on the extent of {A B} satisfying the above properties. First, one constructs, if possible, an equivalence relation ≡ satisfying *structure*, for which A ≡ B. This is done by imitating the method of the very fast (almost linear in time and linear in space) "set union algorithm" analyzed by R. E. Tarjan [14]. This

relation ≡ is then checked for the *no cycles* property by the (linear in time and space) "topological sort" method described by Knuth [9].

The remaining details are too many for a complete discussion in these notes. A recent paper by J. S. Vitter and R. A. Simmons [16] contains a full account, and also discusses the potential speed-up of this algorithm if advantage is taken of the opportunities it offers for parallel computations. Gerard Huet described this idea already in his thesis [7], and there are also papers by Paterson and Wegman [11], and by Martelli and Montanari [10], which give strictly linear algorithms. The following brief sketch will give a feel for this general "set union" approach.

Equivalence classes represented by trees. In the set union algorithm, the classes of an equivalence relation on a set are represented as trees in a forest defined by a set E of equations between distinct elements of the set, no two equations in E having the same left hand side. If E contains the equation U = V we say that E equates U to V, and we think of U and V as being nodes in the same tree, with V the parent of U. The function ROOT is defined for any element A of the set and such a collection E of equations as arguments, and finds the root of the tree in which A lies:

ROOT A E = **if** E equates A to B **then** ROOT B E **else** A.

The equivalence relation $\equiv_E$ represented by E is then that for which

$X \equiv_E Y$ **iff** ROOT X E = ROOT Y E.

The *structure* part of the algorithm is then embodied in two cooperating functions EQUIV and EQUATE. In general the function EQUIV, when given expressions A, B, as its first and second arguments and a collection E of equations as its third argument, returns either E itself, if $A \equiv_E B$, or a set E^+ of equations obtained by adding to E the equations required for E^+ to satisfy *structure* and for $A \equiv_{E^+} B$. If no such E^+ exists, then EQUIV returns the message "FAIL". The function EQUATE has a similar behavior, but assumes that E is *not* the message "FAIL" and that A and B are roots of E.

We then define UNIFY by

(UNIFY A B) = (EQUIV A B { })

where EQUIV and EQUATE are defined by

```
EQUIV A B E  =  if   E is "FAIL" then "FAIL"
                else EQUATE (ROOT A E) (ROOT B E) E

EQUATE A B E =  if   A = B then E                                  1
                else if {A,B} contains a variable then MERGE A B E 2
                else if {A,B} contains a constant then "FAIL"      3
```

else EQUIV **aA aB** (EQUIV **dA dB** (MERGE A B E)) 4

Since EQUATE is called by EQUIV, the input assumption for EQUATE is correct that E is not the message "FAIL" and that A and B are roots of E. The action in line 3 of EQUATE is appropriate to having discovered that *structure* cannot be satisfied, since in view of the tests in lines 1, 2 and 3, one of A, B must be a constant while the other must be either a (distinct) constant or else a cons. The action in lines 2 and 4 is to merge the two trees (i.e. equivalence classes) into a single tree (equivalence class) by equating one root to the other. The action in line 4 is appropriate to having found that both A and B are conses, so that not only are the two trees (equivalence classes) merged, but the resulting forest (equivalence relation) is modified by such further mergings of equivalence classes as may be necessary to satisfy *structure*.

If we use the following simple definition for MERGE

MERGE A B E ≡ **If** A is a variable **then** $\{A=B\} \cup E$ **else** $\{B=A\} \cup E$

then the calls of MERGE from EQUATE preserve the property that

- if a tree representing an equivalence class has a variable as its root, then all expressions in that equivalence class are variables.

Balancing and path compression. *However, this simple method of merging* foregoes the opportunity to imitate fully the set union algorithm, which, when merging, balances the resulting tree by making the root of the taller tree the parent of the root of the shorter tree. The set union algorithm also compresses the trees: each call (ROOT A_0 E) returns a root A_n by accessing a sequence of equations

$$A_i = A_{i+1}, \quad (i = 0, \ldots, n-1)$$

in E (a "path"). Each of these equations is replaced by the equation $A_i = A_n$, making the root the parent of each node A_i, $(i = 0, \ldots, n-1)$ (the path is "compressed"). By complicating the fast unification algorithm slightly to reflect these two refinements, an essentially linear cost performance can be achieved.

Logic. After all these preliminaries we can now get to our main topic and describe the clausal predicate calculus. This is *is a special machine-oriented version of* the predicate calculus, in which one deals principally with only one form of proposition (the clausal sequent) and uses only one inference principle by which to prove the truths among such propositions (the resolution principle).

Clausal sequents. A clausal sequent is an intuitively meaningful proposition which asserts that a given set of universal disjunctive clauses say, the set P, logically entails a given set of existential conjunctive clauses, say, Q.

Sequent notation. We write the sequent by joining the two sets with a sequent arrow, thus:

$P \Rightarrow Q$

and we say that P is the antecedent, and Q the succedent, of the sequent. We read the sequent as "P logically entails Q" or as "Q logically follows from P". If Q happens to be the empty set we can write the sequent simply as

$P \Rightarrow$

and read it as "P is logically unsatisfiable". We often write the antecedent and succedent sets by simply listing their clauses in some order (the order being irrelevant) and omitting the external set brackets. Thus, we write

$A, B, C \Rightarrow M, N$

rather than

$\{A, B, C\} \Rightarrow \{M, N\}$.

Meaning of a clausal sequent. Pending the more exact definitions to be given below, we can say that the sequent $P \Rightarrow Q$ expresses the claim that there is no interpretation of the predicate symbols and function symbols of the clauses in P and Q under which all the clauses in P are true and all those in Q are false.
When Q is a singleton {C} this accords well with our everyday understanding of what it means to say that C logically follows from (the sentences in) P: "C must be true whenever the sentences in P are true".

Universal disjunctive and existential conjunctive clauses. A universal disjunctive (u.d.) clause is a formal sentence of the form:

$\forall x_1 \ldots x_k \; (\neg A_1 \vee \ldots \vee \neg A_p \vee B_1 \vee \ldots \vee B_q)$

which is equivalent to

$\forall x_1 \ldots x_k \; ((A_1 \wedge \ldots \wedge A_p) \rightarrow (B_1 \vee \ldots \vee B_q))$

while an existential conjunctive (e.c.) clause is one of the form:

$\exists x_1 \ldots x_k \; (A_1 \wedge \ldots \wedge A_p \wedge \neg B_1 \wedge \ldots \wedge \neg B_q)$

which is equivalent to

$\exists x_1 \ldots x_k \; ((A_1 \wedge \ldots \wedge A_p) \wedge \neg (B_1 \vee \ldots \vee B_q))$.

In each clause, the expressions $A_1, \ldots, A_p$ and $B_1, \ldots, B_q$ are predications (defined below), and the $x_1, \ldots, x_k$ are all the distinct variables which occur in them. For both kinds of clause we say that

- the set $\{x_1 \ldots x_k\}$ is the prefix of the clause,
- the set $\{A_1 \ldots A_p\}$ is the body of the clause, and
- the set $\{B_1 \ldots B_q\}$ is the head of the clause.

A clause with an empty prefix (i.e. which contains no variables) is called a ground clause.

Empty clauses. When both head and body are empty then (necessarily) also the prefix is empty, and the clause itself is then said to be the empty clause of the one kind or the other. The empty u.d. clause is equivalent to the formal sentence **false**, and the empty e.c. clause is equivalent to the formal sentence **true**. The formal sentence **false** is false in every interpretation (see below), and the formal sentence **true** is true in every interpretation.

The kernel of a clause. A u.d. clause with a given prefix, body and head is transformed into an e.c. clause with the same prefix, body and head under the operation of interchanging $\forall$ with $\exists$, $\wedge$ with $\vee$, and (for each predication P) $\neg$P with P. The prefix, body and head are invariants of the transformation. We shall find it useful to define the basic operations and notions of resolution in terms of these invariants alone. We shall call the triple

$$< \{x_1 \ldots x_k\} \ \{A_1 \ldots A_p\} \{B_1 \ldots B_q\} >$$

consisting of the prefix, body and head of a clause C, the kernel of C, regardless of whether C is a u.d. clause or an e.c. clause. It is convenient to think of the u.d. clause with kernel <X A B> as the tuple <$\forall$ X A B>, and the e.c. clause with kernel <X A B> as the tuple < $\exists$ X A B >. Notice that when a u.d. clause and an e.c. clause have the same kernel then each is logically equivalent to the negation of the other. It has been the usual practice in clausal predicate calculus to deal only with u.d. clauses, and to call these simply clauses without qualification. However, in our opinion this practice leads to unnecessary awkwardness in the later development of the resolution principle. By working with kernels wherever possible we are able to enjoy the conceptual economy of the usual treatment without denying ourselves the richer and more natural means of expression which the availability of both kinds of clause provides.

Predications and terms. Herbrand Universes and Bases. In any particular application, we shall regard the clauses and clausal sequents as all built ultimately out of the variables and the members of a certain set of constants, each constant in which is classified as a predicate symbol or a function symbol and assigned an arity. (The arity of a symbol may be any natural number. When the arity of a function symbol is 0 the symbol is called an individual symbol). Relative to such a fixed set L of predicate and function symbols (sometimes called a lexicon) we define certain expressions to be the terms over L and the predications over L, and when these expressions contain no variables we say they are ground terms over L and ground predications over L. The definitions are:

- The terms over L are the variables, and also the lists $[F\ t_1 \ldots t_n]$ in which F is a function symbol of arity n in L, and the t_i are terms over L. The set of all ground terms over L is called the Herbrand Universe over L.
- The predications over L are the lists $[P\ t_1 \ldots t_n]$ in which P is a predicate symbol of arity n in L, and the t_i are terms over L. The set of all ground predications over L is called the Herbrand Base over L.

Substitutions extended to tuples. It is useful to extend the substitution notation not only (as was done earlier) to sets of expressions, but also to tuples whose components are expressions or sets of expressions, in the obvious way: e.g., the instance of the triple < X A B > by the substitution θ is the triple

< Xθ Aθ Bθ >, and we write it as: < X A B >θ.

We can now readily explain what is meant by a variant of a clause.

Variants. If C is a clause with kernel <X A B> and the substitution θ is a change of variables, then Cθ is also a clause, with kernel <X A B>θ. It is called a variant of <X A B>.

[NOTA BENE: the substitution operates on the variables of the prefix, even though they are from the logical point of view bound variables of the clause. Our substitution operations know nothing about the meanings, if any, that we associate with expressions.]

In particular every clause is a variant of itself (with $\theta = \varepsilon$). Note also that if Cθ is a variant of C then C is a variant of Cθ since

$$C\theta\theta^{-1} = C\varepsilon = C.$$

Finally, if A is a variant of B and B is a variant of C then A is a variant of C. Thus "being a variant of" is an equivalence relation on clauses.

Separated clauses. Instances. Two clauses are said to be separated if their prefixes are disjoint. A clause D is an instance of a clause C if there is a substitution θ such that D = Cθ. (In order for this definition to make sense, the clauses must be construed as tuples). If the prefix of D is empty, it is a ground instance of C.

Herbrand interpretations of L. An Herbrand interpretation J of a lexicon L is given by specifying, for each predication in the Herbrand Base of L, whether it is true in J or false in J.

The idea is behind this way of defining interpretions is that such a specification of truth or falsehood can be automatically extended to clauses, since in every Herbrand interpretation

- the variables in clauses over L range over the Herbrand Universe of L, which is taken as the domain of individuals of the interpretation J; so that
- a u.d. (respectively, e.c.) clause is true (respectively, false) in J if and only if all its ground instances are true (respectively, false) in J; and
- a u.d. (respectively, e.c.) ground clause with body A and head B is false (respectively, true) in J if and only if each member of A is true in J and each member of B is false in J.

Counterexamples of clausal sequents. Let S be a clausal sequent over the lexicon L. Then an interpretation J of L is a counterexample of S if and only if every clause in the antecedent of S is true in J, and every clause in the succedent of S is false in J.

Equivalence of clausal sequents. Two sequents over L are equivalent if and only if every counterexample of either is also a counterexample of the other.

Truth of clausal sequents. A clausal sequent is true if and only if it has no counterexamples.

Being true is in general only a semidecidable property of clausal sequents. There is no general algorithm for detecting the falsehood of every false sequent, but there are, as we shall see, sound and complete systems of proof for clausal sequents, and such a system of proof for clausal sequents is an algorithmic method of recognizing the truth of any clausal sequent which is in fact true. This is what the resolution principle makes possible, in a reasonably efficient form.

Obvious sequents. Some true clausal sequents can be immediately recognized as such: for example, those which contain **false** in the antecedent or **true** in the succedent. We call these obvious sequents. In general, however, a true clausal sequents is not obviously true, and we need some other means of establishing their truth. This is the motivation for the resolution principle.

Proving true clausal sequents by means of resolution. The resolution principle is based upon the following construction, involving a unification algorithm, of a resolvent of the kernels of two separated clauses.

Resolvents. Let E, F be two separated clauses. Let the kernels of E and F be $< X\ A\ B >$ and $< Y\ C\ D >$ respectively. Then a clause R with kernel $< Z\ M\ N >$ is a resolvent of E with F on K if and only if K is a unifiable subset of $B \cup C$ with most general unifier σ, such that both $K \cap B$ and $K \cap C$ are nonempty, and such that

- $Z = (X\sigma \cup Y\sigma) \cap VARS,$
- $M = A\sigma \cup (C\sigma - K\sigma),$
- $N = (B\sigma - K\sigma) \cup D\sigma.$

Note that both the u.d. clause R with kernel $< Z\ M\ N >$ and the e.c. clause R with kernel $< Z\ M\ N >$ are resolvents of E with F on K.

The resolution principle. The resolution principle is an inference principle for clausal sequents. This is a somewhat different point of view from the usual one. In the usual treatment [12], resolution is formulated as an inference principle for u.d. clauses, and allows us to infer a u.d. clause as conclusion from two u.d. clauses as premises. By restating the principle for sequents, however, we uncover more of the power and flexibility of the underlying idea, and also allow e.c. clauses to enter into resolution reasoning in a natural way. So we state the resolution principle as follows:

- from a nonobvious clausal sequent S one may infer
  the clausal sequent S + R, *and conversely*, provided

that R is a resolvent of two separated variants
of clauses in S and provided that R is not a variant
of a clause in S.

By S + R we mean the clausal sequent obtained by adding R to the antecedent of S, if it is a u.d.clause, or to the succedent of S, if it is an e.c. clause. Any such sequent S + R is then said to be a resolution of the sequent S.

Comment on the definition. The provision that R should not be a variant of a clause in S eliminates an undesirable source of redundancy and ensures that a given sequent has only finitely many essentially different resolutions. The provision that the sequent S be nonobvious removes the possibility of continuing to look for a proof when the proof is in fact complete. This will become clear in the next paragraph.

Resolution series; resolution proofs. A series $S_0, \ldots, S_r$ of clausal sequents in which S_{i+1} is a resolution of S_i, $0 \leq i < r$, is called a resolution series. A resolution series is a resolution proof (of its initial sequent) if and only if its final sequent is obvious.

Soundness of the resolution principle. The logical justification of the resolution principle rests on the fact (not difficult to establish) that all sequents in a resolution series are equivalent. This means that all sequents in a resolution proof are true - as is clear from the fact that its final sequent, being by definition an obvious sequent , is (obviously) true. In particular, then, the initial sequent of a resolution proof is true, and so the proof really is a proof of its initial sequent. We simply read the proof backwards, starting with an obviously true sequent, and proceeding in truth-preserving steps until we arrive at the initial sequent with the knowledge that it too must be true. As a proof system for clausal sequents, in other words, the resolution principle is sound. The significance of the resolution principle for computational logic then rests on two further properties of resolution, local finiteness and completeness, one of which is easy to see, the other of which is not.

Local finiteness. A clausal sequent has only finitely many resolutions, all of which can be effectively constructed. Hence we can effectively find all finite resolution series starting with a given sequent, and therefore all resolution proofs of that sequent, if it has any. This fact allows us to build resolution-proof-finding systems.

Completeness of the resolution principle. *Every true sequent has at least* one resolution proof.

This fact is *quite* nontrivial. Because of it, resolution-proof-finding systems are truth-recognition systems for clausal sequents.

Horn sequents. The general resolution-proof-finding procedure as sketched above is rather attractive compared with older proof-finding systems for the predicate calculus. However, it is not yet in an efficient enough form for what we

now call logic programming. There are just too many resolutions at each step for it to be feasible to search out all resolution proofs of a given true clausal sequent. Nevertheless Green [5] was able to use the general procedure to lay out and motivate the main ideas of logic programming as we know them today. Soon thereafter Colmerauer [4] with his PROLOG system, and Kowalski [8] in a more abstract form, showed that by sacrificing some generality one can achieve a remarkably useful system of logical computation. The key idea is to work with a restricted version of clausal predicate calculus rather than the full system. In the restricted system we consider only Horn sequents, rather than clausal sequents in general.

Horn sequents; procedure clauses; goal clauses. A clause is a procedure clause if its head is a singleton, and a goal clause if its head is empty. Both procedure clauses and goal clauses are called Horn clauses. In both kinds of clauses the predications in the body of the clause are called goals. A Horn sequent is a then a clausal sequent whose antecedent contains only procedure clauses and whose succedent contains only goal clauses. A minimal Horn sequent is one which contains only one goal clause.

Logic programs = minimal Horn sequents. A minimal Horn sequent S thus has the form $P \Rightarrow (\exists Y)\ (G_1 \wedge \ldots \wedge G_n)$, where P is a set of procedure clauses, and $\{G_1, \ldots, G_n\}$ is a nonempty set of goals, while the variables in Y are all those which occur in the goals G_i. The various resolution proofs (if any) of S will include the very special LUSH resolution proofs which we define below, but also many, many more. It turns out that if S is true then not only does it have a resolution proof (which we already know) but it even has a LUSH resolution proof. This is crucial for logic programming purposes. It means that a logic programming engine need only search through the very much sparser space of LUSH resolution series starting with a true minimal sequent S, in order to be sure of finding a proof of S.

Selection and removal functions. To help in stating the idea of LUSH resolution we need the idea of a selection function, namely, a function $\uparrow$ defined on every nonempty set M, whose value $(\uparrow M)$ at M is some member of M. To each such selection function corresponds the removal function $\downarrow$, which when applied to a nonempty set M returns the set which is the result of removing the expression $(\uparrow M)$ from M.

LUSH resolution series. Let $\uparrow$ be any selection function. Given the resolution series

$$S_0, S_1, \ldots, S_i, \ldots,$$

let us write

$$S_{i+1} = S_i + R_{i+1} \qquad (i \geq 0)$$

to show that at each step the sequent S_{i+1} is obtained by adding the resolvent R_{i+1} to the sequent S_i. Then the series is a LUSH resolution series controlled by $\uparrow$ provided that

- each S_i $(i \geq 0)$ is a Horn sequent,
- S_0 is a *minimal* Horn sequent with goal clause R_0,
- R_{i+1} $(i \geq 0)$ is a resolvent with R_i, on the set $\{ H_i \uparrow C_i \}$, of (a variant) $\forall X_i \ (A_i \rightarrow H_i)$ of some procedure clause in S_i, where C_i is the body of R_i.

The clause R_i is called the <u>active clause</u> of the sequent S_i. Thus the active clause of each sequent in the series after the first is the resolvent of some procedure clause in the preceding sequent with the active clause of the preceding sequent. The active clause in the initial sequent is the (only) goal clause in that sequent. Note that in a LUSH resolution series the resolvents are always goal clauses: <u>no procedure clauses are generated as resolvents</u>. Note also that each procedure clause can produce <u>at most one</u> resolvent with the active clause, and will do so if, but *only* if, its head can be unified with the goal selected by $\uparrow$ from the body of the active clause. The rapidity with which this can be decided will depend on the complexity of the particular function $\uparrow$ being used, on the method of representation of the sets of goals to which it is applied, and on the method of representing the procedure clauses in the initial sequent. PROLOG uses particularly efficient methods, in which an order is imposed on the goals. PROLOG's $\uparrow$ respects this ordering in the sense that $\uparrow M$ is always the first of the most recently added elements of M. The details are too many for further discussion here.

The idea of LUSH resolution is originally due to Kowalski [8] but the acronymic label is due to Hill [6]. It is intended to suggest: **L**inear resolution with **U**nrestricted **S**election function for **H**orn clauses. The LUSH resolution proofs of a true minimal Horn sequent are far fewer in number than the ordinary resolution proofs of it. Many people, following the example of van Emden and Kowalski [15] and Lloyd [17], prefer to use the acronym SLD (**S**elected **L**inear resolution for **D**efinite clauses) instead of LUSH. However, acronyms are not to be multiplied, or even added, beyond necessity.

LUSH resolution is complete in the following very strong sense: if S is a true minimal Horn sequent, then for <u>all</u> selection functions $\uparrow$ there is at least one LUSH resolution proof of S which is controlled by $\uparrow$.

Lush resolution series as computations. We can associate with the LUSH resolution series $S_0, S_1, \ldots, S_i, \ldots$, controlled by $\uparrow$, the <u>computation</u> which is the series $C_0, C_1, \ldots, C_i, \ldots$, of <u>states</u> each of which is the body of the active clause R_i of the *corresponding sequent* S_i. Consider, then, a LUSH resolution series whose initial Horn sequent $P \Rightarrow (\exists Y)C$ provides the initial state C. The relationship between successive states corresponding to the successive Horn sequents in the series exhibits a repetitive pattern: each successive state C_{i+1} is obtained from its predecessor C_i by the same "computation cycle".

The computation cycle. To obtain a successor of the nonempty state C_i we take some procedure clause in P such that, for some suitable variant

$$<X_i\ A_i\ \{H_i\}>\theta_i\ ,$$

of its kernel $<X_i\ A_i\ \{H_i\}>$, the set $\{\uparrow C_i,\ H_i\theta_i\}$ is unifiable with most general unifier σ_i. If no such procedure clause exists, then the state C_i is a "failure". and has no successors. Otherwise, a new state C_{i+1} is formed by the construction

$$C_{i+1} = A_i\theta_i\sigma_i \ \cup\ (\downarrow C_i)\,\sigma_i\ .$$

In general there may be more than one procedure clause for which this construction can be made. In other words, a state may have more than one successor.

Stacklike behavior of successive states. This computation cycle does indeed abstractly resemble the basic cycle of a simple stack-oriented computer, if we think of $\uparrow$ as returning, and $\downarrow$ as removing, the top element of the stack. The states C_i are then the successive contents of the stack. The cycle thus consists, partly, of popping the top goal from the stack and pushing the goals of the procedure body onto the stack. This is what PROLOG actually does. Unfortunately this simple analogy does not account for the application, to each goal in the stack, of the most general unifier σ_i. However, by means of an idea due originally to Boyer and Moore in their Edinburgh resolution theorem prover [1], we can find a natural computational role in this analogy for the substitutions θ_i and σ_i of this basic cycle: that of the environment of bindings for the variables.

Implicit representation of expressions. The idea of Boyer and Moore is to represent an expression $E\theta$ *implicitly* by the ordered pair $< E\ \theta >$ instead of actually carrying out the work of applying θ to E. This ordered pair can be thought of as a "closure" or a "delayed evaluation". The ordered pair $< E\ \theta >$ can be treated in all respects as though it were the explicit expression $E\theta$ that it implicitly represents: for example the result $E\theta\lambda$ of applying λ to the expression represented by the ordered pair $<E\ \theta>$ is itself represented by the ordered pair

$$<<E\ \theta>\ \lambda>\ ,$$

and so on. When pairs are nested to the left like this we follow an "association to the left" convention and drop the inner brackets. In general, the expression implicitly represented by $< E\ \theta_1 \ldots \theta_n >$ can be found simply by carrying out the "delayed" work of applying the successive substitutions, to yield the explicit expression: $((E\theta_1) \ldots \theta_n)$. It is straightforward to adapt the procedures UNIFY, EQUIV, EQUATE, ROOT, MERGE, etc., to this method of implicit representation of expressions. We can then use the Boyer-Moore idea to represent the successive states of a LUSH computation. Instead of actually explicitly constructing the state C_{i+1} by applying the most general unifier σ_i to the set $(A_i\theta_i \cup \uparrow C_i)$ we can simply

represent C_{i+1} by pairing this set with σ_i:

$$C_{i+1} = < (A_i\theta_i \cup \hat{\uparrow}C_i) \ \sigma_i >.$$

The set $A_i\theta_i$ can also be represented in the same way, so that the equation can be written

$$C_{i+1} = < (<A_i \ \theta_i> \cup \hat{\uparrow}C_i) \ \sigma_i >.$$

Computations on the machine will of course use this implicit representation wherever possible. In particular the successive unification substitutions will be separate components of each state. The successive states of the computation corresponding to a LUSH resolution proof of the minimal Horn sequent

$$P \Rightarrow (\exists Y)C$$

are then the following (in Boyer-Moore form):

$$\begin{array}{lcll}
C_0 & = & < C \ \varepsilon > & \\
C_1 & = & < (<A_1 \ \theta_1> & \cup \hat{\uparrow}C_0) \quad \sigma_1 > \\
C_2 & = & < (<A_2 \ \theta_2> & \cup \hat{\uparrow}C_1) \quad \sigma_1 \ \sigma_2 > \\
 & & \vdots & \\
C_{j+1} & = & < (<A_{j+1} \ \theta_{j+1}> & \cup \hat{\uparrow}C_j) \quad \sigma_1 \ \sigma_2 \ \ldots \ \sigma_{j+1}> \\
 & & \vdots & \\
C_t & = & < \{\} & \quad \sigma_1 \ \sigma_2 \ \ldots \ \sigma_j \ \ldots \sigma_t >
\end{array}$$

with the successive kernels $<X_1 \ A_1 \ \{H_1\}>, \ldots, <X_t \ A_t \ \{H_t\}>$ of (not necessarily different) procedure clauses in P supplying the sets A_j of new goals at each step, and with each substitution σ_j satisfying the equation:

$$\sigma_j = (\text{UNIFY } H_j\theta_j \ (\hat{\uparrow}C_j) < \sigma_1 \ldots \sigma_{j-1} >), \qquad (j \geq 1).$$

Throughout this LUSH computation no expression need actually be constructed explicitly. At termination, the substitution $(\sigma_1 \ \sigma_2 \ \ldots \ \sigma_t)$ is available to construct the output of the computation.

The computation tree. The procedure clause chosen at each step of the computation is one of the only finitely many occurring in the antecedent P of the sequent. This gives rise to a computation space which is a finitary (but not necessarily finite) tree. The various branches of the tree correspond to the various LUSH resolution series starting with the given initial sequent. The root of the tree is the body of the goal clause (= the active clause) of the initial sequent, and in general each node of the tree is either

- empty, and a leaf of the tree, (a "success") or
- nonempty but with no successors, and a leaf of the tree, (a "failure") or
- nonempty and with one or more successors.

The "success" branches (if any) of the computation tree are the completed

computations corresponding to the various LUSH resolution proofs of the initial minimal sequent P ⇒ (∃Y)C the body C of whose active clause (∃Y)C is the root of the tree.

Each completed computation yields an output. It is then natural to view the equation

$$Y = (Y\sigma_1\sigma_2 \ldots \sigma_t)$$

as the output of the completed computation, where the state $< \{\}\ \sigma_1\ \sigma_2\ \ldots\ \sigma_t >$ is its terminal node. The term(s) $Y\sigma_1\sigma_2 \ldots \sigma_t$ are expressions constructed stepwise by the successive unifiers $\sigma_1, \sigma_2, \ldots, \sigma_t$.

Nondeterminacy. For every *true* initial sequent S there is at least one such computation since the sequent has at least one LUSH resolution proof. S may, however, have many, even infinitely many, such proofs; and for the computation corresponding to each proof there will be a possibly different output. It is the purpose of the various logic programming engines, such as a PROLOG machine, to obtain all such outputs when given a true minimal Horn sequent. This it does by making a complete exploration of the search space. If the search space is infinite then it may contain infinitely many success branches (and it may also contain infinitely many failure branches). A properly designed logic programming engine should presumably generate the search tree fairly, i.e., in such a way (and there are many options) as to reach any given node in the tree after only finitely much time. Unfortunately, most PROLOG engines are designed unfairly (for the sake of speed) as "depth-first backtracking" devices.

Example 1. The minimal Horn sequent sequent

- ∀x: [NUMBER x] → [NUMBER [1+ x]]
- [NUMBER 0]

⇒ ∃y: [NUMBER y]

is true, and has infinitely many LUSH resolution proofs.

The outputs of the corresponding computations are the equations:

y = 0
y = [1+ 0]
y = [1+ [1+ 0]]

and so on.

At the jth step (j > 0) the active clause is (understanding y_0 as just y itself):

$\exists y_j$: [NUMBER [1+ . . . [1+ y_j] . . .]] . (with j 1+'s)

and the next resolution can be obtained by choosing either the second procedure clause

(*) [NUMBER 0]

or the [j+1]st variant

(**) $\forall y_{j+1}$: [NUMBER y_{j+1}] → [NUMBER [1+ y_{j+1}]]

of the first procedure clause. The choice of (*) will yield an obvious sequent , and the output will be

y = [1+ . . . [1+ 0] . . .] (with j 1+'s)

The choice of (**) will produce the new active clause

$\exists y_{j+1}$: [NUMBER [1+ . . . [1+ y_{j+1}] . . .]] (with j+1 1+'s).

A depth-first backtracking device which chose (**) before (*) at each level would therefore delay permanently the choice of (*) at the first level, and never reach even the first success node in the search tree. Merely reversing the order of the choice would produce a complete (although of course nonterminating) traversal of the search tree.

Example 2. The two procedure clauses

P1: ∀x: [CAT [ ] x x]
P2: ∀x,a,b,c: [[CAT a b c] → [CAT [• x a] b [• x c]]]

intuitively define [CAT x y z] to mean that the list z is the result of conCATenating the lists x and y in that order. The goal clause

Q: ∃p,q: [CAT p q [• 1 [• 2 [• 3 NIL]]]]

intuitively says that the list [1 2 3] is p ++ q, i.e., the result of concatenating two lists, p and q, in that order. The minimal Horn sequent {P1 P2} ⇒ Q is true, and has four different *LUSH resolution proofs, the simplest of which is obtained* by adding just one resolution invoking the procedure clause P1. The corresponding computation has the output

[p q] = [NIL [1 2 3]]

describing the construction [1 2 3] = [ ]++[1 2 3]. The other three proofs consist respectively of one, two and three successive invocations of the procedure clause P2 followed by an invocation of the procedure clause P1. The outputs of the

corresponding computations describe respectively the constructions

$$[1\ 2\ 3] = [1] {+}{+} [2\ 3],\ [1\ 2\ 3] = [1\ 2] {+}{+} [3],\quad [1\ 2\ 3] = [1\ 2\ 3] {+}{+} [\,].$$

Another view of LUSH resolution proofs. Clark [3] has pointed out an interesting alternative way to view a computation with output $Y = Y\theta$ corresponding to a LUSH resolution proof of a true minimal Horn sequent

$$P \Rightarrow (\exists Y)(G_1 \wedge \ldots \wedge G_n).$$

Namely, it can be interpreted as the construction of n separate hyper-resolution proofs by which n different unconditional procedure clauses

$$(\forall X_1)H_1,\ \ldots,\ (\forall X_n)H_n$$

are simultaneously deduced from the procedure clauses in P, and which are such that the expressions

$$[\,H_1\ \ldots\ H_n\,] \quad \text{and} \quad [\,G_1\ \ldots\ G_n\,]$$

are unifiable with most general unifier θ.

Hyper-resolution. Hyper-resolution is an inference pattern for u.d. clauses which requires a conditional procedure clause

(1) $\quad (\forall X)(A_1 \wedge \ldots \wedge A_k \rightarrow B)$

with $k \geq 1$ goals $A_1, \ldots, A_k$, as major premise, and k unconditional procedure clauses

(2) $\quad (\forall X_1)B_1, \ldots, (\forall X_k)B_k,$

separated from it and from each other, as minor premises, such that the expressions

$$[\,A_1 \ldots A_k\,] \quad \text{and} \quad [\,B_1 \ldots B_k\,]$$

are unifiable with most general unifier σ. The conclusion of the hyper-resolution inference is then the unconditional procedure clause

(3) $\quad (\forall Z)C$

where $C = B\sigma$, and $Z = (X \cup X_1 \cup \ldots \cup X_k)\sigma \cap VARS$. It is straightforward to verify that the clause (3) is indeed a logical consequence of the clauses (2) together with the clause (1).

One may use this inference pattern to obtain, from a set P of procedure clauses, a series $P_0, P_1, \ldots,$ of sets of procedure clauses all of which are logical consequences of P, as follows. The set P_0 is the set of unconditional procedure clauses in P. The set P_{j+1} is the result of adding to the set P_j all the unconditional procedure clauses which can be inferred by a single application of hyper-resolution with major premise in the set P and minor premises in the set P_{j+1}. Clearly the union

$$P^k = P_0 \cup P_1 \cup \ldots \cup P_k$$

of the first k of these sets contains all the unconditional procedure clauses deducible from P by no more than k steps of hyper-resolution. It is natural to organize a deduction from P of such an unconditional procedure clause as a tree with that clause as its root and with members of P_0 as leaves. Each nonleaf node

in the tree is the immediate consequence, by hyper-resolution, of its immediate descendents in the tree (as minor premises) and some conditional clause in P as major premise.

LUSH resolution and hyper-resolution. Now consider again a LUSH resolution proof whose initial sequent is $P \Rightarrow (\exists Y)(G_1 \wedge \ldots \wedge G_n)$ and whose output is $Y = Y\theta$. Let

$$C_0, \ldots, C_t$$

be the successive states of the corresponding computation and let

$$D_0, \ldots, D_t$$

be the successive procedure clauses D_i such that D_i is used with C_i to get C_{i+1}. Note that

$$C_0 = \{G_1, \ldots, G_n\}.$$

Clark observes that the ith step of the computation can be interpreted as attaching the goals in the body of D_i as immediate successors to the goal $\uparrow C_i$. These goals then become, together with those in the set $\downarrow C_i$, the goals in the set C_{i+1}. In this manner n trees are grown, each nonleaf node in which follows, by a single application of hyper-resolution, from its immediate successors (as minor premises) and some procedure clause in P (as major premise). Clark's interpretation is most illuminating, for it shows where the extraordinary freedom of the LUSH resolution scheme (to use any selection function whatsoever) comes from. It comes from the fact that it does not matter in what order the nodes of the n hyper-resolution trees are treated as these trees are grown.

Negation as failure. The preceding discussion has dealt only with the pure Horn clause case of logic programming. In practice, one can work (as first explained in Clark [2]) with pseudo-Horn clauses by generalising the definition of Horn clauses in the following way. Instead of restricting the bodies to be sets of predications (= unnegated atomic sentences) we can allow them to be literals, that is, either predications or negated predications. However, both the definition of procedure clauses as having a head containing exactly one predication, and the definition of goal clauses as having an empty head, are retained. The basic computation cycle is then extended to deal with the case that the goal $\uparrow C_i$ can now be a negated predication, say, $\neg G$ (hence not unifiable with the head of any procedure clause). If G is a ground expression an attempt is made to prove the sequent $P \Rightarrow G$. This attempt can have three outcomes:

1. it can terminate with a proof, in which case $\neg G$ is "disproved" and C_i has no successors
2. it can terminate but without finding a proof ("finite failure"), in which case ("negation by failure") $\neg G$ is "proved" and C_i has the successor $\downarrow C_i$

3 it can fail to terminate, in which case the attempt must be eventually aborted without any decision as to the provability of G from P.

Evidently, **1** assumes the simple consistency of P, while **2** assumes something like the completeness of P ("if G were true it would be provable from P") for ground literals. We cannot discuss further here this important and interesting topic, which is the subject of much current research. We refer the reader to Clark [2] and Lloyd [17] for further details.

References.

[1] Boyer, R.S. and Moore, J S.
The sharing of structure in theorem proving programs.
Machine Intelligence 7, Edinburgh University Press, 1972, 101 - 116.

[2] Clark, K.L.
Negation as failure.
In Logic and Databases, edited by Gallaire and Minker,
Plenum Press, 1978, 293 - 322.

[3] Clark, K.L.
Predicate logic as a computational formalism.
Ph.D. Thesis, Imperial College, London, 1979.

[4] Colmerauer, A., et al.
Un systeme de communication homme-machine en francais.
Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Universite
d'Aix-Marseille, Luminy, 1972.

[5] Green, C.C.
Application of theorem proving to problem solving.
Proceedings of First InternationalJointConference on
Artificial Intelligence, Washington D.C., 1969, 219 - 239.

[6] Hill, R.
LUSH resolution and its completeness.
DCL Memo 78, University of Edinburgh Department
of Artificial Intelligence, August 1974.

[7] Huet, G.
Resolution d'equations dans les langages d'ordre 1, 2, . . . , ω.
These d'Etat, Universite Paris VII (1976).

[8] Kowalski, R.A.
Predicate logic as programming language.
Proceedings of IFIP Congress 1974, North Holland, 1974, 569 - 574.

[9] Knuth, D. E.
The Art of Computer Programming, Volume 1, Addison-Wesley 1969,
258 - 268.

[10] Martelli, A. and Montanari, U.
Unification in linear time and space.
Technical Report B76-16, University of Pisa, Italy, 1976.

[11] Paterson, M. S., and Wegman, M. N.
Linear unification. Journal of Computer and Systems Sciences
16, 1978, 158 - 167.

[12] Robinson, J. A.
A machine-oriented logic based on the resolution principle.
Journal of the Association for Comp[uting Machinery 12, 1965, 23 - 41.

[13] Robinson, J. A.
Automatic deduction with hyper-resolution.
International Journal of Computer Mathematics 1, 1965, 227 - 234.

[14] Tarjan, R. E.
Efficiency of a good but not linear set union algorithm.
Journal of the Association for Computing Machinery 22, 1975, 215 - 225.

[15] van Emden, M.H., and Kowalski, R.A.
The semantics of predicate logic as a programming language.
J.ournal of the Association for Computing Machinery 23, 1976, 733 - 742.

[16] Vitter, J. S., and Simmons, R. A.
New classes for parallel complexity: a study of unification
and other complete problems for ***P***.
I.E.E.E.Transactions on Computers, Vol C-35, May 1986, 403 - 417.

[17] Lloyd, J. W.
Foundations of logic programming.
Springer-Verlag, 1984. Second, extended edition, 1987.