PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

Logic Programming Schemes

Keith L. Clark

Department of Computing
Imperial College
180 Queens Gate
London SW7 2BZ

Abstract

This paper offers a tutorial but incomplete survey of a succession of proposed logic programming language schemes all of which can be considered variants or descendants of the original Kowalski scheme. Semantic properties and implementation issues are discussed.

1 Introduction

It is now 14 years since Kowalski(1974) described his scheme for logic programming based on a backward chaining resolution inference system for Horn clause programs and 15 years since the first implementation of Prolog (Battani & Meloni 1973), a language which is an instance of that scheme. Since, many extensions of the Kowalski scheme have been proposed, and many successors of Prolog have been implemented. This paper is a tutorial introduction to a succession of logic language schemes starting with the Kowalski scheme and ending with the CLP scheme of Joxan and Lassez (1987).

The schemes are presented using a common framework derived from Colmerauer(1982). The semantic properties are discussed using concepts from (Clark 1979,1982). This was the first formulation that focussed on the semantics of the answer substitutions returned by the Kowalski scheme, interpreting them as conjunctions of equalities denoting relations over the Herbrand universe. This is the formulation that best relates the Kowalski scheme to the other schemes, which compute descriptions of relations represented by conjunctions of more general formulas, possibly over other domains.

Schemes not based on first order classical logic, such as lambda-Prolog (Miller and Nadathur 1986) and schemes based on bottom up evaluation (Ramakrishnan 1988) (see also Bancilhon & Ramakrishnan 1986) are not covered.

2 Units of computation

The unit of computation of many logic programming schemes is the unification of one or more pairs of terms. Terms are constructed from countably infinite disjoint alphabets F, V of *functors* and *variables*. Each functor has an associated arity. Functors with arity 0 are *constants*. A *term* is a variable, a constant, or of the form f(t1,..,tk), k≥1, where f is a k-adic functor. $t(F,V)$ is the set of all terms and $t(F)$ is the set of ground (variable free) terms.

An *assignment* is a set A of equations {X1=t1,..,Xn=tn} where Xi are distinct variables and ti are terms (which may contain variables). ti is called the *binding* for Xi. A *substitution* S is an assignment where no Xi occurs in any ti. A *unifier* Ø of a pair of terms t,t' is a substitution such that t{Ø} is identical to t'{Ø}. t{Ø} is t with any variable bound by S is replaced by its binding. We give below an algorithm for computing a unifier of a set of pairs of terms given as a set of equations. The algorithm actually returns a most general unifier (mgu). See (Robinson 1979) or (Lassez et al 1987) for a more formal definition of unifier and mgu.

The difference between an assignment and a substitution is important. Let (E)S denote the existential closure of S, the existential quantification of all its variables. (E)S is true no matter what interpretation I is given to the functors. Such an interpretation is a pair $<D,A>$ where D is a non-empty domain and A is a function mapping each k-adic functor into a k-adic function from D^k to D. Let M be a function mapping the variables Y1,..,Yk, of the binding terms in S to elements e1,..,ek of D. Letting Xi = v(ti), the value of ti for interpretation A,M gives a tuple of values X1=v(t1),..,Xn=v(tn),Y1=e1,..,Yk=ek which trivially satisfies the equations S.

If S is just an assignment, (E)S is not always true for every I. Consider the assignment {X=f(X)}. (EX){X=f(X)} is true only if f has a fixed point.

Herbrand interpretation

An interpretation for which (EX){X=f(X)} is false is the free or Herbrand interpretation HI. The domain is the set

the functors has a special role in logic programming. A substitution denotes a non-empty relation for any interpretation because it denotes a non-empty relation in the Herbrand interpretation.

Unification algorithm

The following algorithm finds a unifier of a set of pairs of terms expressed as set of equations E={t1=t'1,..,tk=t'k}. The algorithm terminates with a substitution S (a success termination) or it terminates with a set of equations containing **false**, (a fail termination).

(a) Replace any equation of the form f(t1,..,tk)=f(t'1,..,t'k) by the k equations t1=t'1,..,t=t'k.
(b) Delete any equation of the form X=X.
(c) Replace any equation of the form t=X, t a non-variable, by X=t.
(d) Select any equation of the form X=t, where X occurs in some other equation and X≠t.
  (i) If X occurs in t (the occur check), terminate replacing the equation by **false**.
  (ii) Otherwise, replace X by t in all other equations. X=t i is not deleted .
(e) If there is an equation f(...)=g(...), where f and g are different, terminate replacing the equation by **false**.

The above algorithm always terminates. If it fails, the original set of equations E has no unifier. If it terminates with success then S is a substitution which is an mgu of E. A proof is in (Martelli and Montanari 1982). The algorithm is essentially Herbrand's original algorithm for checking whether or not a set of equations has a solution for the Herbrand interpretation (HI) of its functors.

Rule d(ii) can be modified to the instruction: replace X by t in some of the other equations. The rule can then be reapplied, possibly to a different binding equation X=t' for X, until X does not appear in any other equation. This is a modification of the algorithm used in the some of the parallel languages we shall discuss in section 4.

Let us call an equation of the form X=t to which d(ii) applies a binding equation. Applying d(ii), is *communicating* or *broadcasting* the binding. Applying the substitution to every equation is *global* broadcasting, applying it to only some equations in which X appears, is *local* broadcasting. In an implementation in which a variable is a pointer to a single memory location, from which any binding is automatically retrieved, global broadcasting is achieved by simply assigning the value t of a binding equation X=t to the location for X. Local broadcasting must be implemented by having multiple locations for X, or by explicitly copying and substituting for X in those equations covered by the local broadcast.

An implementation that stores bindings instead of substituting in equations corresponds to the following modification of rule (d).

(d') Select any equation of the form X=t.
If there is a broadcasted equation X=t' for X, replace X=t by t'=t. Otherwise,
  (i) If X occurs in t (the occur check), terminate replace the equation by **false**.
  (ii) Otherwise, make X=t the broadcasted equation for X.

Local broadcasting to some subset of equations is the further modification of this rule to incorporate the idea of a broadcast equation restricted to certain other equations. The test then checks for a broadcast equation for this eqaution.

We can also allow simultaneous selection of several equations, each to be handled by application of the appropriate rule, providing we impose some restrictions. If we have global broadcasting, only one binding equation can be selected for any variable. If we have local broadcasting, we can select multiple binding equations providing each is not included in the broadcast range of any other.

Unification as a special form of equation solving

Let X1,..,Xk be the variables of the original set of equations E. E can be be viewed as defining a relation R_E={<X1,..,Xk>:E} given some interpretation I. An empty set of equations defines {<>:**true**}, or equivalently just the logical constant **true**. The equations have a solution for I iff R_E is not empty for I. The final set S produced by the unification algorithm is such that R_S is contained in RE for any interpretation I. If S is a substitution, this tells us that R_E is non-empty, hence that the equations have a solution, for any interpretation I. If S contains **false**, R_S is the empty relation **false**. We cannot conclude that R_E=**false** for any I, but we can conclude that is empty for the interpretation HI. This is because each of the equation rewrites preserves equivalence for HI. The free interpretation of the functors is reflected in rules (a), d(i) and (e).

Unification as inference from a theory of the functors

The following axioms, from (Clark 1978), characterise the Herbrand interpretation.

(F1) for every functor
  f(X1,..,Xk) = f(Y1,..,Yk) -> X1=Y1,..,Xk=Yk

(F2) for every pair of distinct functors f,g
f(X1,..,Xk) ≠ g(Y1,..,Yn)
(F3) for every non variable term t(X) containing some variable X, X≠t(X)

These axioms, together with the normal reflexive, symmetric, transitive and substitution axioms for =, comprise the Herbrand equality theory HET. Each step of the unification algorithm is a particular use of one of the axioms of the equality theory. This relationship is formalised in the result (Clark 1978):

(R2.1) HET |= R_E=R_S, or equivalently V(E <-> S)
where V denotes universal closure.

That E <- S only needs the general equality axioms, which establishes that R_E contains R_S for all interpretations. That E -> S needs the freeness axioms.
As corollaries of (R2.1) we have
(R2.2) S=**false** iff HET |= R_E=**false**
(R2.3) S is a substitution iff R_E≠**false** for any I

Thus S is equivalent to E for theory HET and a simple test on the syntactic form of S tells us whether or not E has a solution. S is a *solution form* for equations E for the theory HET.

Solving equations for other theories of the functors

The unification process can be viewed as a special case of a more general process of checking whether or not a set of equations has a solution for some other equality theory. (Colmerauer 1982) is an example of a logic programming scheme in which equations are solved for a different theory, an equality theory of infinite rational trees.

Checking solvability of more general equality formulas

Viewed as a description of a relation, a set of equations is a just a special case of a first order formula using only the predicate =. Unification could be replaced by a process of checking whether or not some more general equality formula is satisfiable (denotes a non-empty relation) for some interpretation of the functors. Colmerauer's (1986) scheme is an example of this. In this scheme, the formulas are conjunctions of equalities and negated equalities reduced to solution form for a theory of infinite rational trees.

Checking solvability of constraint formulas

The final generalization, is to allow other predicates in the formulas for which we need to determine satisfiability. The key logical requirement is that there is some theory T of the functors and predicates that appear in the formulas which is *satisfaction complete* for the formulas. That is, for any formula it determines whether or not the formula denotes a non-empty relation. Checking this property for some formula F, is this generalization of unification of Jaffer and Lassez's(1987) scheme for constraint logic programming.

Pragmatic requirements

Testing satisfiability of a set of equations using unification is suitable unit of computation for a logic programming language because it is algorithmic, it produces an equivalent solution form and the algorithm is incremental. To reduce a larger set E∪E' to solution form we can apply the algorithm to S∪E' where S is the solution form for E. Moreover, the solution form form for S∪E' it is expressible as an extension S∪S' of S if rule (d') is used instead of rule (d). This incremental nature of the algorithm is essential for efficient implementation. This is because a unification based logic program computation is essentially the process of reducing to solved form progressively larger sets of equations.

In the non-unification based schemes, we must also have 'algorithmic' and incremental reducibilty to solved form if we are to justify the *programming* label.

3 Schemes based on unification

3.1 SLD resolution - the Kowalski scheme

The first schematic framework for logic programming languages was given by Kowalski (1974). This scheme is actually LUSH resolution (Hill 1974), now referred to as SLD-resolution. Programs comprise rules of the form

A <- A1,..,An for n≥0

where A, A1,..,An are atoms. An *atom* is of the form p(t1,..,tn), n≥0, where the ti are terms and p is a n-adic predicate taken from some countably infinite set *P* of predicate names disjoint from *F* and *V*. A is the clause *head* and A1,..,An are *calls* comprising the clause *body*. It is a clause *about* r, if r is the predicate of A. Each variable in the clause is implicitly universally quantified.

A computation is invoked by a conjunction G of calls. If X1,..,Xk are all the variables of G, it is a request to a description of one or more instances of the relation R_G={<X1,..,Xk>:G} for any interpretation which is a model of the program.

A state of the computation is a pair <G,S> where G is a conjunction of calls B1,..,Bm and S is a substitution or it contains **false**. In the initial state S is the empty substitution {}. A unit of computation is the unification of some call Bi= r(t'1,..,t'k), selected by a *computation rule* CR, with the head A=r(t1,..,tk) of a clause variant A<-A1,..,An (a clause with variables renamed so as to have no variables in common with G) in the binding environment represented by S. This is the application of the above unification algorithm to $S_U\{t1=t'1,..,tk=t'k\}$, or equivalently to $S_U\{t1=t'1,..,tk=t'k\}\{S\}$ to produce a solution form S'. A next state of the computation is

<**true**,S'> if m=1,n=0,otherwise

<B1,..,Bi-1,A1,...,An,Bi+1,..,Bm,S'>.

The alternative states that can be derived using different clauses for the predicate of the selected atom represent branch points for the computation. The *computation tree* for a goal G and computation rule CR is a finitely branching tree routed at <G,{}> labelled with states. The offsprings of a node in the tree are all the alternative states that can be generated using all the clauses about the predicate of the selected call.

A branch of the computation tree terminates with *success* if its end node is labelled by a state <**true**,S> where S is a substitution. A computation path terminates with *failure* if its end node is labelled with a state <G,S>, S contains **false**. A branch may be infinite.

The strategy for constructing the computation tree is the *search* strategy. The strategy is *fair* if it does not indefinitely postpone the construction of some branch of the search tree.

The answer computed by a success branch on a tree for computation rule CR is the substitution S restricted to bindings X1,..,Xk for the variables of the original goal clause G. (If rule unification rule d' is used instead of d, S must also be applied to the bindings for X1,...,Xk.) The answer S' can be interpreted as denoting a k-adic answer relation $R_{S^\wedge} = \{<X1,..,Xk> : S'^\wedge\}$, where S'^ is the existential quantification of S' with respect to all the variables in the binding terms. Important theoretical results, strengthening the soundness and completeness results of (Hill 1974), were proved in (Clark 1979). They are:

(R3.1.1) *Soundness*

For every CR-computed answer substitution S,

P |= R_G contains $R_{S^\wedge}$

Since for a substitution, $R_{S^\wedge} \neq$ **false** for any I, this result implies the usual but weaker soundness result :

If there is a success computation path then P|=(E)G

(R3.1.2) *Independence of the computation rule*

For each CR-computed answer S there is an CR'-computed answer S' such that $R_{S^\wedge}=R_{S'^\wedge}$ for every I.

(R3.1.3) *Strong completeness*

For every substitution S' such that P |= R_G contains $R_{S'^\wedge}$ there is an CR-computed answer S such that $R_{S^\wedge}$ contains $R_{S'^\wedge}$ for every I.

Proof of (R3.1.3) uses the fact that the unification algorithm returns an mgu but I believe it can be proved using only (R2.1).

The Kowalski scheme allows for or-parallel search down the alternative evaluation paths but only and-sequential evaluation. At each computation step only one call is being unified. A trivial extension, is to allow concurrent evaluation of any sequence of k steps from state <G,S> that select atoms which can be independently unified with heads of clauses in the context S. The condition that guarantees this is that each pair of atoms B, B' of the sequence of selected atoms are such that B{S} has no variables in common with B'{S}. This is *independent and-parallelism.*

Another simple extension, is to allow the return of qualified answers. Suppose the original goal has been reduced to a state <G',S'> where S' is a substitution but G'≠**true**. Let S be the subset of bindings in S' that bind variables in G. A qualified answer (Vasey 1986) for that computation path is S, G' . A simple generalization of (R3.1.1) tells us that

P |= R_G contains $R_{(S,G')^\wedge}$

Of course, $R_{(S,G')^\wedge} \neq$ **false** only if P |= (E)G'

Types of computation rule

A rule that always selects one of the introduced calls A1,..,An if there is one, is a *depth first* rule. The rule that always selects the leftmost introduced call, or the next call in the goal conjunction if there are no introduced calls, is the *leftmost call* rule. A rule that does not always select an introduced call is a *coroutining rule*. With a coroutining rule the computation can alternate between the evaluation of different calls. With a depth first rule calls are always completely evaluated once selected but the calls are not necessarily selected in the order in which they appear in the goal and in the body of clauses.

Implementations of the Kowalski scheme

The first implementation was Prolog (Battoni and Meloni 1973), which actually predated the publication of the

Kowalski scheme. Prolog uses the leftmost call computation rule. It also uses a depth first backtracking search strategy trying the clauses in the fixed order in which they are entered.

IC-Prolog (Clark and McCabe 1979, Clark et al. 1982) was the first implementation to allow more general computation rules specified by program annotations. Like Prolog it uses depth first backtracking search. The default rule is the leftmost call rule but a different order of the calls in the body of a clause can be specified for different modes of use, causing the calls to be introduced into the goal in different orders for different modes. A mode of use is a restriction of the unification with the head of the clause which specifies certain argument terms as input or output. Suppose that we successfully unify a call r(t1,..,tk) in in state <G,S> with a head r(t'1,..,t'k). The input restriction on t'i is satisfied if all the bindings that result from rewriting equation ti{S}=t'i are for variables in t'i. The output restriction on t'j is satisfied if tj is a variable that is not bound in S.

A coroutining rule in IC-Prolog is specified by annotations on variables in calls. A ? annotation on a variable V in a call, B, specifies B as a eager consumer of the binding for V. Suppose the leftmost call rule selects a call A to the left of B and the unification with the next clause for the atom results in the broadcasting of a non-variable binding t for V. The body atoms for the clause are introduced into the goal but call B is moved from its position in the current goal conjunction G to the leftmost position. The evaluation then continues with a depth first evaluaton of B. However, no call B', which is a descendant of B, is allowed to broadcast a non-variable binding for a variable in t. If the unification of such a call B', would result in the broadcasting of such a binding, the call is not selected. Instead, the conjunction of all the current descendants B1,..,Bj of B are moved back to the position that B occupied in goal G. The computation continues with the new leftmost call. The descendants of B are moved to the front of the goal if a non-variable binding for any variable in t is computed and back again to the position they occupied if the selection of one of the descendants would result in the broadcasting of a non-variable binding for a variable in t'. The switching forward and back continues until there are no descendants of B. The consumer annotation has no effect if no call to the left of B tries to bind V. A dual notion, that of an lazy producer, is specified by the annotation ^ on a variable in a call B. Strict one step alternation between the depth first evaluation of two or more calls can also be specified, giving pseudo parallel evaluation.

A weaker form of the eager consumer concept of IC-Prolog, which is easier to implement, was independently devised by Colmerauer and colleagues and implemented in Prolog II (Colmerauer 1982b). This is the freeze call. Instead of annotating the variable V in a call B with ^, the call is written freeze(V,B). Suppose now that the leftmost call computation rule selects freeze(V,B). If V is bound to a non-variable in the current environment S, the evaluation continues as though the call was B. If V is unbound, B is temporarily removed from the goal conjunction of the current state and linked with V. Other freeze calls can add to the number of frozen calls linked with V, as can binding V to some other variable U that has linked frozen calls. Now suppose that a non-variable binding for V is broadcast by the unification of some call B' with the head of a clause A<-A1,..,An. B and all other frozen calls linked with V are reintroduced into the goal in front of the body atoms A1,..,An. The freeze condition is not inherited, a descendant of B will resuspend only if it is contained inside another freeze call.

If V is never bound to a non-variable, B and any other frozen calls linked with V will never be re-introduced into the computation. So this implementation computes qualified answers, the qualification being the conjunction of all frozen calls that are not reintroduced. IC-Prolog does not compute qualified answers because all calls always remain in some position in the goal.

In MU-Prolog (Naish 1985) coroutining is also implemented by temporarily removing the leftmost call from the goal. Here, the suspension condition is not specified by the form of call, but by the specification of allowed modes of use of clauses. With each predicate r a set of modes of use is specified by a set of *wait* statements. A *wait* statement specifies a subset of argument positions that are allowed output positions for the unification of any call for r with the head of each clause for r. When the left most call B is for a predicate with *wait* statements, the call is suspended if the unification with the next clause for r would result in the broadcasting of bindings not allowed by any *wait* statement for r. Let U1,..,Uj be those variables for which bindings cannot be broadcast without violating some *wait* statement. The call B is removed from the goal but reintroduced at the front of the goal as as soon as a non-varible binding is broadcast for any of these variables. The call may resuspend.

In Chapter 3 of Naish (1985) an algorithm for automatically generating *wait* statements is given. These

have the effect of suspending any call for which a depthfirst evaluation would result in an infinite computation branch.

In NU-Prolog (Thom and Zobel 1887), *wait* statements are replaced by *when* statements which give conditions under which a call can be selected rather than conditions for suspension. The *when* declaration is dummy clause for a predicate that must succeed before any call to the predicate can be selected. Its role is to check that certain arguments are of a particular form, or are non-variable, or are ground terms. As in MU-Prolog, they can cause a call to be temporarily removed until one or more variables are bound to non-variable terms.

(Ciepielewski & Haridi 1984) and (Moto-Oka et al 1984) are designs for or-parallel implementation of SLD with a leftmost call rule. (Conery 1987) has or-parallelism, restricted and-parallelism and dynamic re-ordering of the body calls depending upon which variables in the body are bound by the head unification. (Degroot 1984) is a proposal to allow restricted and-parallelism in an otherwise sequential leftmost call implementation with compile time analysis used to simplify the runtime test for independence of calls. Warren (1987) is a survey of current work on the or-parallel implementations of Prolog.

3.2 Heterogeneous SLD - the Naish scheme

An interesting variation of the SLD, called Heterogeneous SLD, has been proposed by Naish(1984) as a more suitable model for a coroutining implementation. In this variant, it is still the case that only a single call is unified and replaced by the body atoms of some clause during a computation step, but the successor states do not all have to be produced by replacing the same selected call. Some of them can be produced by selecting another call and using only *some* of the clauses for the predicate of that call. The computation rule applied to a state returns a sequence of <call,clause> pairs to be used in generating the successor states, although the sequence order does not constrain the order in which some search strategy generates the successor states. A state of the computation also records the clauses that can be still used to unify with a particular call. Suppose the computation rule returns a sequence <Bi,C>,<Bj,C'>,.. for the state <G,S>, and <Gi,Si> is the successor generated by unifying Bi with the head of C and <Gj,Sj> is the successor obtained by unifying Bj with the head of C'. Then, C' will appear in the list of clauses that can be used to unify with the occurrence of Bj in Gi, but clause C will not appear in the list that can be used for Bi in Gj.

Naish proves that providing the sequence returned by the computation rule includes all the (to be used) clauses for at least one call, the scheme computes the same set of answers as SLD. Operationally, it allows a backtracking implementation which has discovered that all computation paths that result from using clause C to try to solve call Bi in <G,S> ends in failure, to backtrack to <G,S>, select another call Bj in G, and subsequently to try to solve Bi without needing to reconstruct the failure subtree generated by using C. This is because C will have been deleted from the clause set associated with Bi down the new branch routed at <G,S>. Naish also points out that it justifies the following form of intelligent backtracking: once a call has been found to fail, on backtracking to the parent state, retry the failed call if it is in the goal rather than the previously selected call. Repeat this until the call succeeds or until it is no longer in the backtrack goal.

IC-Prolog and MU-Prolog are implementations of this scheme. This is because the attempted unification which causes a call B to be delayed may be an attempted unification with the second or later clause for B. When B is resumed, the previous clauses are not retried.

3.3 SLDNF - the Clark scheme

In (Clark 1978) an extension of SLD was proposed to allow negated atoms in queries and the bodies of rules. Programs are now expressed as implications of the form

A <- L1,...,Lk

where A is an atom and each Li is a literal : an atom B or its negation ~B. Goals are conjunctions of literals. Each variable in a rule is still implicitly universally quantified.

States of the computation are as in SLD except that the goal component is a conjunction of literals and it may contain **false**. The computation rule SR can select any positive call or any negative call ~B in <G,S> such that B{S} is ground (variable free). Following (Lloyd and Topor 1986) we shall call such a rule *safe*. If the selected call is positive, this is a normal call, and is handled as in SLD.

If the selected call is a negative call ~B then the query evaluation process is recursively entered with the ground query B{S}. If every computation path ends in failure, ~B is assumed to succeed and is deleted from the current goal to give the new goal. If some computation branch for B{S} ends in success, ~B is replaced by **false**, giving a failure termination. This is the *negation as failure* rule.

A computation branch *flounders* if a state is generated

which only has an unground negative calls.

Prolog is an SLDNF systems but without the safety constraint on the computation rule. It is left to the programmer to make sure the negated calls come after positive calls that can be used to generate ground bindings for their variables. MU-Prolog and NU-Prolog have safe computation rules. IC-Prolog has an unsafe rule but raises an error if the evaluation of a negated call tries to broadcast a binding for a variable in B{S}.

The answers computed by SLDNF are not logical consequences of the program. Firstly, failure to unify is interpreted as proof of falsity, which is only valid for the Herbrand interpretation. Secondly, there is an implicit assumption that the given clauses somehow constitute a complete definition for each relation.

Program completion

(a) r(t1,..,kn) <- L1,..,Lk

is a program clause. The *guarded form* of the clause is

(b) r(X1,..,Xn) <- X1=t1,..,Xn=tn : L1,..,Lk

and the *general form* is

(c) r(X1,..,Xn) <- (EY1,..,Yj)
(X1=t1,..,Xn=tn : L1,..,Lk)

X1,..Xn are new variables not in (a) and (EY1,..,Yj) is the existential quantification of all the variables in (a). The ':' is just a conjunction connective like ','.

Remember that unification between a selected call r(t'1,..,t'n) and the clause head r(t1,..,tn), is the unification of $E_U\{t'1=t1,...,t'n=tn\}$. This is equivalent to the unification of $E_U\{X1=t'1,..,Xn=t'n\}_U$ X1=t1,..,Xn=tn}. If we revised our definition of a computation step, so that all guard equations are also added to E before it is reduced to solution form, computation using the guarded form of the clause is exactly the same as computation using the original clauses. Let

r(X1,..,Xn) <- E1
r(X1,..,Xn) <- E2

r(X1,..,Xn) <- Em

be the general forms of all the clauses for r.

The *completed* definition for r is:

r(X1,..,Xk) <-> E1 $\vee$ E2 $\vee$ .. $\vee$ Em.

Program completion

completed(P) comprises:

(i) the completed definition for every predicate that appears in the head of a clause in P,

(ii) the definitions

Q(X1,..,Xk) <-> false k≥o

for every k-adic predicate for which there are no program clauses.

comp(P) = completed(P)$_U$HET

Soundness results for SLNDF (Clark 1978)

(R3.3.1) For every SR computed answer S for goal G,
comp(P) |= R_G contains $R_{S^\wedge}$

(R3.3.2) If for some safe computation rule every branch of the computation tree for G ends in failure then comp(P) |= ~(E)G

(R3.3.3) If the SR-computation tree routed at G is finite and S1,..,Sn, n≥0, are all the answers computed by the success branches, then

comp(P) |= $R_G = R_{S1^\wedge} U$ ..... $U R_{Sn^\wedge}$, equivalently
comp(P) |= (V)[G <-> S1^ $\vee$.....$\vee$ Sn^]

Allowing unshared local variables in the negated call

(R3.3.2) will allow us to generalize SLDNF programs to have negated conjunctions ~C, not just negated calls. We can liberalise the safety condition and allow ~C to be selected if C{S} does not share a variable with any other call in G{S}. These local variables are then implicitly existentially quantified inside the negation because a completely failed computation is a proof of ~(E)C{S}. This is what happens in the Prolog implementation of negation as failure. We can have a rule

maths_major(X)<-
student(X),~(maths_course(Y),~takes(X,Y))

The evaluation of maths_course(Y),~takes(X,Y), with X=p, , will be a proof of

~(EY)(maths_course(Y),~takes(p,Y))

if it fails. Unfortunatly, this relaxation does effect the simple rule that all variables of a clause are implicitly universally quantified and it is better to insist that the negated condition is explicitly existentially quantified in the rule as in NU-Prolog (Naish 1986). NU-Prolog(Naish 1986) enforces the generalized safety check that all unquantified variables of such a negation (its *global* variables) are bound to non-variables before it can be selected. The quantified variables are its *local* variables.

Suspending evaluation of negated call

Another extension of the negation as failure rule would be to allow selection of a quantified conjunction ~(EV$_L$)C, with unbound global variables V$_G$, but to suspend any branch of the computation of C{S} that tries to broadcast a non-variable binding for any of these variables. The branch is resumed when a binding for the variable is broadcast. The evaluation will coroutine between the recursively entered goal C{S} and the evaluation of the original goal G. Suppose there is a successful evaluation of C{S} that succeeds without binding *any* global variable, we can replace ~(EV$_L$)C by **false**. For we have established (VV$_G$)(EV$_L$)C{S} and hence ~(EV$_G$)~(EV$_L$)C{S}. No matter what ground bindings are given to the variables in V$_G$, C{S} will succeed. IC-Prolog has this generalization of the rule, but it raises an error instead of suspending if there is an attempt to bind a global variable.

Allowing generation of values for global variables

(R3.3.3) will allow a negation as failure rule which generates answers. Suppose we allow any negated condition ~C to be selected and that we always try to construct the complete computation free for C{S}. If there is a success branch that does not result in any bindings for variables in C{S}, we can replace C{S} by **false**. If all branches fail, we can delete it. If S1,..Sn are all the answer bindings for variables in C{S}, then result (R3.3.3) tells us that ~C can be replaced by the conjunction ~S1^,...,~Sn^. Distributing the negation will give us quantifed negated equalities. To handle these, we need to extend the SLDNF scheme so that unification is replaced by the process of checking that such inequalities are consistent with the binding equations returned as answers to other calls. We shall return to this in section 6.

Constructive proof

As pointed out in (Clark 1978), SLDNF requires each negated atom to be constructively proven, it does not allow case analysis proofs. Thus, q is a consequence of the completion of the program

p <- p q <- p q<- ~ p

but the evaluation of q will not terminate under any computation rule. This is because the evaluation cannot make use of the law of the excluded middle, p V ~p, hence it cannot show that q is true no matter what the the truth of p is.

Completeness results

Unfortunately there is no simple completeness result. The problems are threefold. Firstly, there is no guarantee that for an arbitrary program that the computation will not flounder, terminate with only non-ground negated calls in the goal. Secondly, it must be the case that everything that can be inferred from comp(P) can be inferred 'constructively', without using the law of excluded middle. Thirdly, when there is a 'constructive' proof of ~B{S} from comp(P), we must be able to generate a finite failure tree. The first two conditions force us to put extra constraints on the syntactic form of P. The last one requires that the computation rule be *fair* as well as safe.

A *fair* computation rule is a concept introduced into logic programming by Lassez and Maher. It is a computation rule which does not indefinitely postpone the selection of any call. No depth first computation rule is fair, a fair rule must be a coroutining rule. The following program from (Clark 1978) is an example of a program and goal that require a fair computation rule:

p(X) <- q(Y),r(Y)
q(h(Y)) <- q(Y)
r(g(Y))

With Prolog's leftmost call rule a single branch infinite computation tree is generated for call p(a). With any fair rule it is finite.

Hierarchical programs

Consider the directed graph representing the relation *refers to* for the predicates of P. There is a +(-) labelled edge in the graph between predicates p and q if q appears in a positive(negative) atom in some clause for p. A edge can be labelled with both + and -. In a *hierarchical* program, there can be no cycles in this graph. Note that this rules out recursion. In (Clark 1978) there was some informal discussion of completeness for *hierarchical* programs. The key result concerning hierarchical programs was later given by Sherpherdson (1985):

(R3.3.4) If the program is hierarchical, every variable in a clause occurs in a positive literal in the body, and every variable in a negative literal in a goal G occurs in a positive literal, then for every SLDNF answer S' such that comp(P) |= R$_G$ contains R$_{S'}$^ there is an SR-computable answer S such that HET |= R$_S$^ contains R$_{S'}$^

The conditions concerning variables ensure that the evaluation will not flounder and the hierarchical condition

ensures that every call generates a finite computation tree, hence that everything can be inferred from comp(P) constructively.

As (Lloyd and Topur 1986) showed, the syntactic condition of program clauses can be slightly relaxed for predicates that are only used in negated calls. For the clauses for these predicates only the local variables of the body of the clause have to appear in positive atoms, it is not necessary for variables in the head to appear in a positive atom. Programs and goals that satisfy the Lloyd and Topur conditions are called *allowed*. For allowed programs and goals every computed answer is a set of ground bindings for the variables of the goal.

Completeness for negation free programs

A completeness result that is of considerable importance, even though it is only the base case of a general result, was given by Jaffer et al (1983):
(R3.3.5) For pure definite clause programs (i.e. programs that do no contain negated atoms in the bodies of clauses), when comp(P) |= ~B. for some ground atom B then for every fair computation rule every branch of the computation tree for B ends in failure.

One might hope to use this result to allow negated calls in allowed programs which are calls to predicates defined by negation free clauses. But as soon as we allow this, we can have non-constructive proofs from comp(P). It allows the program

p <- p q <- p q <- ~p

which has the negated call ~p defined by a negation free program.

Completeness for strict programs

A completeness result that constrains P so that there *cannot* be a non-constructive proof from comp(P) is given by Kunen(1988b) for *strict* programs. A program is not *strict* (Apt et al. 1988) iff in its *refers to* graph there are is a pair of relations p,q such that there is a path from p to q which contains an even (possibly 0) number of - labelled edges and a path from p to q that contains an odd number of - labelled edges. In a strict program a predicate p cannot be defined directly or indirectly in terms of positive and negative atoms for some predicate q. The Kunen result is:
(R3.3.7) Suppose P and G are allowed and P is strict. If comp(P) |= GS for some ground substitution S, then S is an SLDNF computable answer. If comp(P) |= ~EG, then there is a finitely failed SLDNF tree for G.

Sherpherdson (1988) and Kunen(1988) are both excellent recent surveys of many other results concerning the semantics, soundness and completeness of negation as failure, for the concept seems to have aroused a lot of interest.

4 Parallel unification based schemes

4.1 GLD resolution -Wolfram, Maher, Lassez scheme

The first scheme to allow unrestricted and-parallel evaluation - the concurrent unification of two or more calls with shared variables is the (Wolfram et al. 1984) GLD scheme. Programs and goals are as in SLD but a state of the computation is a pair <G,S> where G is a multiset of calls. As in SLD, S is a substitution or contains **false**. The computation rule selects n≥1 calls {B1,..,Bn} from G. If {A1<-G1,..,An<-Gn} are variants of n program clauses (with no variables in common with G) where Bi and Ai have the same predicates, then a next state of the computation is

$\langle G \cup G1 \cup ... \cup Gn, S' \rangle$

where S' is the unification solution form of

$S \cup \{B1=A1,..,Bn=An\}$.

(We assume that the unification algorithm is trivially extended to handle the rewriting of r(t1,..,tk)=r(t'1,..,t'k) using rule (a) where r is a predicate.)

Asynchronous GLD

In the GLD scheme, the scope for parallelism is limited because the entire unification must terminate before goals can be selected for the next step. The reduction of each selected goal to the body of a clause is a synchronized step. The following generalization of GLD, implicitly described by Wolfram at al, allows for asynchronous reduction of calls on different processors.

Asynchronous GLD (AGLD) has states <G,E> where G is a multiset of calls and E is a multiset of equations which may contain **false**. The computation rule selects k≥0 calls B1,..,Bk from G and n≥0 equations from E, k+n≠0, where each equation is such that one of the unification rules applies. Each selected equation is handled using the appropriate rule. Let E' be the rewritten set of equations. Let A1<-G1,..,Ak<-Gk be k variants of clauses for the predicates of the selected calls. Let S be the subset of E' that are bindings that have been, in this or a previous step, globally broadcast to all other equations in E. Then

$<G \cup G1 \cup \ldots \cup Gk, E' \cup \{B1\{S\}=A1,..,Bk\{S\}=Ak\}>$
is a next state of the computation. The definitions of success and failure and computed answer are as for SLD and the results of the Wolfram et al. paper show that the computed answers are exactly the same as SLD for every computation rule.

Types of computation rule

GLD is AGLD with a computation rule that always selects only equations to rewrite until the E component of the state has been reduced to solution form. At the other extreme, we can have a computation rule that only selects equations when the goal component of the state is **true**, delaying all unification until the end of the computation. Wolfram et al. use this rule to prove the independence of the computation rule, which is now just a way of collecting together the final set of equations to be reduced to solution form.

Consider the intermediary computation rule which allows selection of atoms along with equations but which delays the selection of any of the introduced atoms Gi from the body of the i'th clause until Bi{S}=Ai has been reduced to a set of binding equations Si, and which only locally broadcasts these bindings to equations descended from Bi{S}=Ai. That is, bindings for shared variables of Bi are not initially communicated to the other concurrently selected calls. Moreover, body calls in Gi are then unified relative to the locally extended $S \cup Si$. This corresponds to an implementation in which each call is evaluated as in SLD with the computed bindings for shared variables of concurrently selected calls being compared (by global broadcasting of the bindings) only when each call has been reduce to **true**. The Epilog system of (Wise 1986) and the Prism system of (Kasif at al 1983) are or-parallel versions of AGLD with this computation rule.

A sequential AGLD rule is one which at each step selects a single equation or a single atom to replace. If it always selects equations until E is in solution form, this is the same as SLD. An intermediary sequential rule generalizes SLD because it allows coroutining implementations to do partial unification, which is not undone, before switching to another atom. For example, a call Bi that is not allowed to generate a non-variable binding for X can be selected and the unification rewrite of Bi{S}=Ai pursued until a binding X=t is generated. This binding is retained, but not broadcast. The next call, or the designated producer of X is then selected in order to generate a binding X=t' which is broadcast. The unification rewrite of the Bi{S}=Ai can then continue with t'=t in place of X=t.

Absys (Foster & Elcock 1969), which can with justification be considered the first logic programming language (see Foster 1988), is essentially an implementation of AGLD with a sequential rule. Terms are restricted to variables, constants and lists and programs are entered in a syntactic variant of the completed definitions of 3.3. The computation rule never selects an equation of the form X=Y, X and Y distinct, for application of rule d(ii). Such variable/variable equations remain as qualifications to computed answers if no other equation rewrite generates a non-variable binding for X or Y. Absys also implemented the negation as failure rule but without the safety check.

Data flow parallel rules

The generalization of a sequential coroutining rule that selects another atom when a binding equation for some *input* variable of the call is generated, is a parallel rule which delays the selection of any call in the body Gi of the clause Ai <- Gi, until Bi{S}=Ai has been reduced to a set of *allowed* bindings, bindings that can be globally broadcast. Any binding X=t made for some designated *input* variable X of the call, is not an allowed binding. Such a binding cannot be broadcast. The handling of X=t is suspended until a binding equation X=t' is otherwise generated and broadcast to the equation X=t.

If need be, we can distinguish between occurrences of variables, preventing broadcasting of a binding equation only if it is generated by rewriting some particular term in Bi. Let us call such variable occurrences, however specified, *input variable occurrences* for the call Bi. Only binding equations generated for non-input variables of Bi are allowed, and can be globally broadcast.

Delaying the global broadcast of allowed bindings

Allowed bindings generated by the rewrite of Bi{S}=Ai do not need to be broadcast immediately they are generated. Data flow rules are used to delay the evaluation of the call until certain bindings have been broadcast to it because the appropriate clause to use for the evaluation of the call is to be determined by the form of the received bindings. This is why we delay the selection of any call in Gi until the unification of Bi{S}=Ai, in the context of all the extra broadcasted bindings it receives, succeeds. If we delay broadcasting any allowed binding until Bi{S}=Ai has been completely reduced to allowed bindings, an attempted unification that fails will have no effect on the evaluation of any other call. We can with impunity substitute for the clause Ai <- Gi some other clause A'i <- G'i such that Bi{S}=A'i does reduce to a set of allowed bindings. We

will usually need to locally broadcast allowed bindings to other equations descended from Bi{S}=Ai, for example to check for incompatible bindings for variables in the clause head, but we can delay the global broadcasting to all other equations in the environment E. Note that delaying the global broadcasting of some allowed binding Y=t for a variable means that Y=t may be transformed into t'=t if some allowed binding for Y generated by another call is globally broadcast. The unification rewrite of Bi{S}=Ai only terminates when all its allowed bindings have been globally broadcast.

Atomic unification

Suppose there is a state of the AGLD computation in which Bi{S}=Ai has been completely reduced to a set of allowed bindings Si (possibly after the receipt of globally broadcast bindings for some variables in Bi{S} generated by other unifications). If the computation rule is constrained so that if it selects one of the bindings from Si for global broadcasting, it must select them all, the rule implements *atomic unification*. (As remarked in section 2, we must already constrain the rule so that it selects only one binding for any variable for global broadcast.)

If, in addition, no call in Gi is selected before all the allowed bindings generated from Bi{S}=Ai have been selected for global broadcasting, the rule implements *atomic test unification*.

In a multiprocessor implementation, atomic unification requires synchronization of the global broadcasting of variables bindings. A given processor must get binding permission from all other processors that might generate an alternative allowed binding for each of the variables bound in Si before broadcasting the bindings. It must be prepared to relinquish the binding permissions given if it cannot get binding permission on them all. A quite complex interprocess protocol is therefore needed to implement atomic unification on a multiprocessor.

Specifying the allowed bindings for a call

The analogue of the freeze call of Prolog II is some sort of annotation in call Bi on the input variable occurrences. Concurrent Prolog (Shapiro 1983) does this by annotating the input occurrences with ?. As with the freeze call, the restriction is not inherited. Occurrences of variables in a binding t' received for an ? annotated variable of a call Bi are not also input variables of Bi, unless they are also annotated with a ?. If the input variable property was automatically inherited, and in addition applied to all calls descended from Bi, this would be the analogue of the IC-Prolog eager consumer.

An alternative way to determine the allowed bindings for the unification of the call Bi{S} with some clause head Ai, is to associate a allowed mode of use with the clause, as in MU-Prolog.

In the Relational Language and Parlog(Clark & Gregory, 1981,1986), this is done by specifying for each argument for each k-adic predicate r an input ? or output ^ mode. Let r(t'1,..,t'k) be the head of the clause being tried and r(t1,..,tk) the call. If i is in an input argument position, then only bindings generated for variables in clause head term t'i are allowed bindings for the unification of ti{S}=t'i. If the unification rewrite generates an equation V=t for any other variable, this is treated as an input occurrence and the binding cannot be broadcast. If j is an output argument position, all bindings generated by the unification of tj{S}=t'j are allowed. The rewriting of the equations for the output argument terms is started only after all the input argument equations have been reduced to allowed bindings. Note that this means that all bindings broadcast to a suspended equation V=t for input variable V must be generated by the unification of other calls. It also means that all variables in the input argument term ti{S}, and all variables in bindings for these variables globally broadcast before t'i=ti{S} is reduced to only allowed bindings, are input variables of the call. The input property is inherited.

In GHC (Ueda 1985), for the whole unification of Bi{S}=Ai only bindings for variables in the clause are allowed. In Parlog terms, every argument position is input. All output in GHC is done by explicit equality calls in the body of the clause.

Guard calls

The global broadcasting of allowed bindings for the unification Bi{S}=Ai is always delayed until there are no disallowed bindings. In addition, we could further delay their global broadcasting until some *guard* subset G'i of the body calls in Gi have been reduced to **true**. Note that during the evaluation of the guard subset we must also prevent global broadcasting of allowed bindings, having only local broadcasting to equations generated by the evaluation of the guard calls and the rewrite of Bi{S}=Ai. In GHC this is done by inheriting the input variable restriction, all input variables of the call are input variables for the all the calls in G'i and their descendants. In Concurrent Prolog, only local broadcasting of call variables is allowed until the guard successfully terminates. In Parlog, only calls that cannot generate allowed bindings for

the input variables of the call can appear in the guard. Such a guard is called *safe*. In the *flat* versions of all three languages, only calls to primitives are allowed in the guard set G'i. Evaluating these calls can then be implemented as an extension of the unification with the clause head.

Note that this holding back of the broadcasting of allowed bindings for call variables means that we can in parallel unify with the clause heads and evaluate the guard calls of all the clauses for Bi. These parallel evaluations will not compete for the broadcasting of bindings. It also means that we can commence the guard calls as we commence the rewrite of Bi{S}=Ai.

Committed choice

Suppose that there is a state of the computation in which Bi{S}=Ai and evaluation of all the guard calls G'i have successfully terminated producing a set of allowed bindings Si. In the Relational language (the first committed choice language) and in Parlog and GHC, there is a commitment to use the clause Ai <- Gi for call Bi. All competing parallel unifications and guard evaluations using other clauses for the call are aborted, immediately, or on termination of the unification and the guard calls. Calls in Gi-G'i can be selected, and, in the Relational Language and Parlog, the rewriting of the equations t'j{S'}=tj{S} for the output argument terms is commenced. The broadcasting of the allowed bindings generated by these rewrites is not atomic, they can be broadcast independently as and when they are generated. In GHC, selection of equations in Gi-G'i will generate these allowed bindings for call variables, which are also not atomically broadcast.

In Concurrent Prolog, there is no commitment to the clause Ai <- Gi and no call in Gi-G'i is selected, before all allowed bindings generated for call variables during the rewrite of Bi{S}=Ai and the evaluation of the guard calls are atomically globally broadcast.

In Parlog and GHC one must program in such a way that only one call will generate a binding for each shared variable, with all other calls suspending until that binding is broadcast. In Concurrent Prolog, one can allow calls to compete, with the atomic test unification making sure that only one binding is globally broadcast *and* that calls are forced to test the broadcast value before committing to a clause. The disadvantage is the complexity of the implementation of atomic unification.

(Burt & Ringwood 1988) have recently proposed a simpler notion of atomic test broadcasting of a single allowed binding as an extension to Flat Parlog. A single allowed binding for a call variable is designated as the test binding. The computation rule must select this designated allowed binding and globally broadcast it before trying to globally broadcast any other binding or select a Gi-G'i call.

A recently proposed successor of Flat Concurrent Prolog, the language FCP(|,:,?) (Klinger at al 1988), borrowing ideas from (Saraswat 1988), divides the guard calls into an *ask* component and a *tell* component. Only the *tell* component can generate allowed bindings for variables not in the clause, for the unification of the call with the head of the clause and the evaluation of the *ask* component only bindings for clause variables are allowed. The *tell* component has a role similar to the unification with the output argument terms in Parlog, for no allowed binding generated by the *tell* component is broadcast to the head unification or the *ask* calls. The difference is that in FCP(|,:,?), there is no committment to the clause until the head unification and the guard succeed *and* all the allowed bindings of the *tell* component have been atomically broadcast.

(Takeuchi and Furakawa 1986) and (Shapiro 1988) both survey the family of committed choice concurrent logic languages based on the AGLD scheme, with examples of programming techniques.

Suspending until only one clause will unify

An alternative computation rule, is to select any call Bi for which there is only one clause with a head Ai which will unify with Bi{S}. To implement this, the different clauses must be tried with local broadcasting of bindings. If more than one call/head unification has been reduced to a set of bindings, the call is suspended until bindings are broadcast from elsewhere which cause all but one unification to fail. All the bindings generated by that unification can then be broadcast. P-Prolog (Yang & Aiso 1986) has has such a suspension rule.

Parallel selection until all calls suspend

In any language which has suspension of calls waiting for variable bindings to be broadcast, deadlock can arise. One can break the deadlock, by picking a single call and ignoring the suspension rules.

In Andorra Prolog (Brand et al 1988), deadlock is broken in just this way. The computation rule selects any number of calls providing there is only one candidate clause for the call, calls suspend when there is more than one clause. A candidate clause for a call is one for which the head unification succeeds generating only allowed bindings and some set of guard calls to primitives successfully

terminates. The allowed bindings for a call are specified by *wait* declarations similar to the *when* statements of NU-Prolog. A commit operator can make a clause the only candidate clause clause, as in the committed choice languages. No binding is globally broadcast until a single clause remains as candidate. The language has atomic test unification. If all calls are suspended, due to wait declarations, or because there is more than one candidate clause for each call, a single call is selected and alternative new states of the computation are generated for each candidate clause to be pursued as or-parallel computations. The language combines the search capability of the SLD scheme with the concurrency of AGLD with committed choice. The penalty is a more complex implementation than is required by either extreme.

P-Prolog also has both or-parallel and and-parallel evaluation but the or-parallel forking does not delay until all the and-parallel calls suspend. It has uncommitted and-parallelism with parallel evaluation of the alternative computation paths.

The CP language of Saraswat (1987) has committed and uncommitted and-parallelism with the concept of a call block. A call block limits the broadcasting of bindings to calls and their descendants in the block. The bindings are broadcast between sibling blocks only when each call in the block successfully terminates. Putting each call in its own block, gives the communication on termination computation rule we mentioned above that is used in Epilog and Prism. Saraswat's language also allows both parallel and sequential (backtracking) search of the alternative evaluation paths.

Parallelised NU-Prolog(Naish 1988) and ANDOR-II(Takeuchi et al 1987) are other recent proposals for mixing committed choice and-parallelism with uncommitted exploration of alternative evaluation paths. (Clark and Gregory 1987) is a discussion of ways in which Prolog and Parlog might be combined.

5 Schemes based on general equation solving

5.1 Removing the occur check - Colmerauer's equation solving over rational trees

Nearly all the implementations of the SLD or SLDNF schemes do not implement the occur check, rule d(i), of the unification algorithm. In (Colmerauer 1982) this 'bug' was turned into a feature. In his scheme, the *rational tree* scheme RTS, programs and goals are as in SLD but answers are assignments and states of the computation are pairs <G,A> of goals and assignments. Remember that in an assignment we can have an equation X=f(a,X) which is not allowed in a substitution.

Colmerauer's equation solving algorithm

Rules (a),(b), (c) and (e) are the same as in the unification algorithm. Rule (d)(i) is deleted, as we would expect. Rule d(ii) becomes two rules, which distinguish two cases covered by d(ii):

(dii1) Select any equation of the form X=Y where X and Y are distinct variables. If X occurs in other equations, replace all other occurrences of X by Y. X=Y is not deleted.

(dii2) Replace any pair of equations of the form X=t1,X=t2, where X is a variable and t1,t2 are not variables and $|t1| \leq |t2|$, by the pair X=t1,t1=t2. $|t|$ is the number of occurrences of elements from $F \cup V$ in t.

Rule (dii2) limits the application of the replacement of a variable by its non-variable binding. This is necessary to ensure termination. With the old formulation of d(ii), we would not terminate when the system contains a pair of equations such as X=f(Y),Y=g(X) because of the absence of d(i).

What is the relationship of the solved form A produced by this algorithm to the original set E. For the unification algorithm we have the result (R2.1). For Colmerauer's algorithm we must delete (F3) from the freeness axioms in HET to produce the rational tree equational theory RTET. We then have:

(R5.1.1) RTET $\models R_E=R_A$

Hence

(R5.1.2) A=**false** iff RTET $\models R_E =$ **false**

However, we cannot also conclude

A is a assignment iff RTET $\models R_E \neq$**false**

because an assignment does not denote a non-empty relation for every interpretation. We must strengthen RTET with axioms that tell us that every assignment does denote a non-empty relation. Following (van Emden and Lloyd 1984), the simple way to do this is to add the axiom scheme (E)A , A any assignment, to RTET to give a set of axioms IRTET. IRTET is a first order theory of the infinite rational trees described in Colmerauer (1982). Such a tree contains a finite number of sub-trees but some of the sub-trees can be infinite. In this domain, the assignment X=f(a,X) has a solution, which is the infinite rational tree f(a,f(a,f(a,.....))). We have:

(R5.1.3) A is a assignment iff IRTET |= $R_E \neq$ false

Correctness of the Colmerauer scheme

(R5.1.4) (van Emden and Lloyd 1984)

For every RTS computed assignment A for goal G,

IRTET,P |= R_G contains R_A, $R_A \neq$ false

Independence of the computation rule and a result analogous to the strong completeness result for SLD should also apply.

Extending RTS to include the negation as failure rule for safe computation rules is straightforward. Since the proofs of (Clark 1978) rely only on the use of the completed definitions and the analogue of (R5.1.1), they should with slight modification apply to RTS. In the soundness results we replace Comp(P) by $RTET_{\cup}completed(P)$. Appropriate versions of the completeness results of section 3.3 should also apply.

5.2 Equation reduction using an general equality theory

Jaffer et al (1986) present a general scheme in which the set of equations introduced at each computation step are reduced to a substitution using an inference from a general equality theory *E*. The scheme is a generalization of GLD, which we shall call GLDE. Instead of unification using the Herbrand unification algorithm, the unit of computation is finding an *E*-unifier of a set of equations E. The analogue of property (R2.1) of unification is a property they call *unification completeness* that E must satisfy.

Let t and t' be two terms. They are *E*-unifiable if there is a substitution S such that $E \models \forall(S \rightarrow t = t')$. Generally, there will be many *E* unifiers, possibly an infinite number. There may or may not be maximally general unifiers, the analogue of the mgu. Even if there are, there may be more than one maximally general unifier. The equivalent of (R2.1) for a general equality theory *E* is that

$E \models \forall(t=t1 \leftrightarrow S1 \vee \ldots\ldots \vee Si \vee \ldots\ldots)$

where $S1 \vee \ldots\ldots \vee Si \vee \ldots\ldots$ is a disjunction of all the *E*-unifiers of t,t'. The <- follows by definition of an *E*-unifier. The -> is the condition of interest. If this condition holds, *E* is said to be *unification complete*. When there are no *E*-unifiers, unification completeness tells us that

$E \models t \neq t'$

the property we need to justify negation as failure.

In GLDE, program clauses are implications of the form

A <- E:B

where E is a conjunction of equality atoms and the B is a conjunction of non-equality atoms. A is a non-equality atom. A goal is the same form as a clause body.

A state of the computation is a three-tuple of multisets <E,B,S> where each E is a multiset of equality atoms, B is a multiset of a non-equality atoms and S is a substitution or **false**. The computation rule selects a subset E'={e1,..,em} of E, a subset B'={A1,..,An} of B, m+n≠0. Let

A'1 <- E1:B1 ..... A'n <- En:Bn

be n variants of program clauses. Let S' be an *E*-unifier of $E \cup \{A1\{S\}=A'1,..,An\{S\}=A'n\}$. That is,

$E \models \forall(S' \rightarrow E \cup \{A1\{S\}=A'1,..,An\{S\}=A'n\})$

A next state of the computation is

$<E-E' \cup E1 \cup .. \cup En,\ B-B' \cup B1 \cup .. \cup Bn, S\{S'\} \cup S'>$

If there is no such S', the next state has **false** in place of the substitution component. A computation branch terminates in success when E,B are both empty and S is a substitution. As Jaffer et al. remark, in any instance of the scheme, maximally generally unifiers should be used if they exist.

Taking *E* to be the empty equality theory, we get GLD as an instance of this scheme. S' is then the mgu that can be generated by the unification algorithm. Jaffer et al. (1986) prove the following soundness and completeness result:

(R5.2.1) If A is a ground atom, $P,E \models A$ iff there is a successful computation for goal A using any computation rule.

(R5.2.2) If A is a ground atom, completed(P),$E \models \sim A$ iff for a fair computation rule every branch of the computation ends in failure.

Note that the second result is not the exact analogue of (R3.3.5) because the failure computation tree can be infinite. This is because there can be an infinite number maximally general unifiers of a set of terms, and so the computation tree may not be finitely branching. If we know that there are always only a finite number of maximally general *E*-unifiers, it is the analogue of (R3.3.5).

The importance of the scheme is that it is a very general framework in which two crucial properties of a logic programming language, (R5.2.1) and (R5.2.2), have been shown to hold. Any instance of the scheme, proposed as a *programming* language, must also have extra computational properties. There must be an algorithm that can be applied to the set of equations

$S \cup E \cup \{A1=A'1,..,An=A'n\}$

that returns a finite set of maximally general *E*-unifiers.

6 Schemes based on testing solvability of more general equality formulas

6.1 Prolog II - Colmerauer's equation, inequation scheme for rational trees

In (Colmerauer 84) the equational solving algorithm for the RTS scheme was extended to apply to sets EI of equations and inequations. An inequation is a negated equality t≠t'. The algorithm divides EI into the set E of equations and the set I of inequations. E is reduced to an assignment A={X1=t1,..,Xn=tn} or **false** using the algorithm of the RTS scheme. If an assignment is generated, for each inequation t≠s in E the algorithm is reapplied to the $A_{\cup}\{t=s\}$. If this produces **false**, the inequality t≠s is discarded (because it is satisfied by assignment A). If the algorithm successfully terminates without generating any bindings for variables, t≠s is replaced by **false** (because the absence of bindings for variables Y1,..,Yk in t=s shows that t=s is satisfied for assignment A for all rational tree values for Y1,..,Yk, hence that there is no rational tree assignment for these variables that will satisfy t≠s and equations A). If s=t is reduced to a set of bindings Y1=t'1,..,Ym=t'm for variables in t=t', then t≠t' is replaced by the the inequation one(Y1,..,Ym) ≠ one(t'1,..,t'm) (because one(Y1,..,Ym) ≠ one(t'1,..,t'm) iff for some i Yi≠t'i iff t≠t'). The result of the algorithm is therefore either **false** or $A_{\cup}I'$ where I' is a set of reduced inequations. Note that no variable bound in A will appear on the left hand side of an inequation in I'. The theory RTET justifies this algorithm, we have:
(R6.1.1) RTET |= $R_{EI} = R_{A,I'}$

If neither A nor I' reduce to **false**, $A_{\cup}S'$ is what Colmerauer calls a *reduced form* set of equations. He shows that a reduced from set of equations always has a solution in the domain of infinite rational trees.

In Colmerauer (1986) this extended algorithm replaced the unification step of a scheme that is the theoretic model for Prolog II. Programs essentially comprise implications of the form

H <- E,I : G

where H is an atom, G is a conjunction of atoms and E is conjunction of equations and I a conjunction of inequations. The rule is still read as universally quantified for every variable. A goal, has the form of a clause body.

A state of the computation is a triple <G,A,I> where G is a conjunction of atoms, and $A_{\cup}I$ is a reduced form set of equations and inequations, or contains **false**.

The first step of a computation is the reduction of the E,I of the goal to solution form, i.e. to reduced form or **false**. Thereafter, a computation step is the selection of some atom Bi in G using the computation rule. If H<-E",I":G" is a clause for the predicate of the selected atom Bi, a next state of the computation is <G',A',I'> where G' is G with Bi replaced by the conjunction G", A is the solution form of $A_{\cup}E"_{\cup}\{Bi=H\}$ and I' is the reduced set of inequations produced by applying the above algorithm to each inequation in $I_{\cup}I"$ using the new assignment A'. The computation terminates in failure if either A or I become **false**. It terminates in success, if G=**true**, and neither A nor I contains **false**. A,I is the computed answer, it will be a reduced form set of equations and inequations.

The theory IIRTET, which is RTET augmented with an axiom scheme (E)A,I , A,I any reduced form set of equations and inequations, gives us the correctness result:
(R6.1.2) If $A_{\cup}I$ is a computed answer for goal E,I,G and program P then P,IIRTET |= $R_{E,I,G}$ contains $R_{A,I}$ and IIRTET |= $R_{A,I} \neq$ **false**

In an implementation of the scheme, the inequations can actually be handled by a special negation as failure rule that returns bindings. After the assignment A' for the new state has been generated, only the new inequations in I", and any inequations in I for which the left hand side variable Y has become bound in A', need be tested. For each such inequation, s≠t, an attempt is made to establish s≠t by trying to show that s=t fails in the environment A'. If s=t fails, s≠t is deleted. If s=t succeeds, without binding any variables in s or t, s≠t is **false**. If it succeeds generating bindings Y1=t'1,..,Ym=t'm for variables in the equation. We have proved that, for theory RTET,

A' -> [s=t <-> (Y1=t'1,..,Ym=t'm)]

The Yi bindings are undone (locations assigned to Y1,..,Ym have there values reset to undefined) and

one(Y1,...,Ym) ≠ one (t'1,..,t'm)

is returned as the 'answer' for the negated call s≠t. This is a single inequation representing the disjunction $Y1 \neq t'1 \vee ... \vee Ym \neq t'm$ which we have proved to be equivalent to s≠t in the environment A'.

6.2 SLDCNF - the Chan Scheme

In 3.3 we hinted that result (R3.3.3) could be used to allow negated calls to return answers. The handling of inequations in the above scheme is a special case of this. The SLDCNF scheme of (Chan 1988) handles answers returned any negated call of a SLDNF style program.

As in the above Prolog II scheme, a computation step involves checking whether a set of equations and inequations has a solution, but for the theory HET underlying normal unification. Inequations can be universally quantified for some or all their variables. As with Prolog II, the inequations are checked for solvability by applying a specialized negation as failure rule. An inequation (VL)s≠t is **true**, and deleted from the current set of inequations, if s=t fails in the environment of the current binding equations. It is replaced by **false**, if it succeeds without binding global variables (variables not in L). (This is one of the extensions to the negation as failure rule we discussed in 3.3.) Unlike Prolog II, the inequation is not reduced to another inequation if s=t succeeds. In this case, (VL)s≠t is considered a *primitive* inequation and left unchanged.

Program rules and goals are as in Prolog II except that the inequations can be universally quantified for some (or all) of the variables in the inequation and body calls can be negated as in SLDNF. A state of the computation is of the form <G,E,QI> where G is a conjunction of calls, E is a set of equations and QI is a set of quantified inequations. E or QI can contain **false.** As with AGLD, we let S be the substitution subset of E of globally broadcast bindings.

A computation terminates in success when G=**true**, E is a substitution S, and QI is a set of primitive inequations PI. Since each primitive inequation is satisfiable in the environment of the substitution S, the set S,PI denotes a non-empty relation for theory HET. The computed answer is a *normalised* form for S',PI, where S' is S restricted to variables in the goal G. S',PI is normalised by a two step transformation to produce an HET equivalent answer. Essentially, this process removes irrelevant inequalities (see Chan 1988). In a normalised form each variable in an equation, and each free variable in an inequation, is either in G or it appears inside a constructed term of some binding term in an equation. So, where there are no function symbols in the bindings, every variable in a normalised answer is a variable from G.

Chan also gives a procedure which converts the negation of a normalised answer N into a disjunction E1∨....∨En, where each Ei is a conjunction of equations and quantified inequations. This is needed to handle the answers returned by a negated call ~B. The negation as failure rule of SLDCNF, recursively evaluates B{S} even if it contains unbound global variables. If every computation path is finite, a finite set {N1,...,Nk} of normalised answers is returned. By an extension of (R3.3.3), the evaluation has shown that B{S} <-> N1^ ∨...∨ Nk^, so ~B{S} <-> ~N1^,...,~Nk^. Each Ni is converted into a disjunction E1∨....∨En and then the whole conjunction for ~B is converted into disjunctive normal form. The result is an equivalence ~B <-> BE1∨...∨BEn where each BEi is a conjunction of equations and quantified inequations.

The computation rule for SLDNF can select any call Bi in the goal G, any equation in E to which a rule of the unification algorithm applies, or any inequation in QI of state <G,E,QI>. The rule does not need to be safe.

A selected inequation is tested to see if it is **true** or **false** in the environment S by the special negation as failure rule described above. If neither, it is left in QI.

If a positive call Bi is selected, let A <- E",QI":G" be a clause for its predicate. A next state of the computation is <G',E',QI∪QI> where E' is the set of equations produced by applying the rules of unification algorithm to E∪E"∪{Bi=A} until Bi{S}=Ai has been reduced to a set of bindings or **false.** In this step, no equation in E∪E" is selected, but it might be changed by the broadcasting of bindings produced from Bi{S}=Ai.

If a negative call ~B is selected from G, a recursive computation for goal B{S} is commenced. If every path is finite, a set of normalised answers {N1..,Nk} is returned. This is negated and converted into an HET equivalent set of {BE1,...,BEn} representing the set of answers to ~B consistent with the bindings S. Suppose BEi is of the form Ei,Qi where Ei is a set of equations. A next state of the computation is <G',E∪Ei,QI∪QIi>.

The correctness result given by Chan is:
(R6.2.1) If every branch of the computation tree is finite, and N1,..,Nk are the set of normalised answers for its success terminating branches, then comp(P) |= (V)[G <-> N1^∨...Nk^].

He gives no completeness results. It should be possible to prove completeness for restricted classes of programs as for SLDNF. Certainly completeness should obtain for hierarchical programs.

(Kunen 1987) gives an alternative approach to allowing negated calls to return answers based on the manipulation of

what he calls *elementary* sets. But at the time of writing I could not see how to present his scheme as an extension of SLD.

7 Constraint schemes

7.1 The CLP scheme - Joxan and Lassez

Joxan and Lassez (1987) present the most general scheme yet proposed that is an extension of SLD. It is a generalization of the scheme we discussed in 5.2. The equality theory E becomes a constraint theory C and the unification completeness property of E becomes a *satisfaction completeness* property of C. The following is a slight generalization of the variant of the CLP scheme given in (Maher 1986), which better fits the framework of this paper. SLD, AGLD, AGLDE and Prolog II are special cases of this CLP scheme.

The theory C is a theory for a set of constraint predicates P_C (disjoint from the set of program predicates P) which includes =. A primitive constraint is an atom with a predicate from P_C. An allowed constraint is some subset of all the first order formulas that can constructed from the primitive constraints which minimally contains all equations and is closed under conjunction. For SLD, C is HET and only conjunctions of equations are allowed constraints. For Prolog II, C is IIRTET, and conjunctions of equations and inequations are allowed constraints.

Theory C must be *satisfaction complete* for the allowed set of constraints. That is, for every allowed constraint C, we have

$C \models (\mathbf{E})C$ or $C \models \sim(\mathbf{E})C$

This is the generalization of properties (R2.2) (R2.3) of HET. IIRTET has this property for the allowed constraints of Prolog II.

Programs comprise implications of the form A <- C : G where C is an allowed constraint, A is a program atom, G is a conjunction of program atoms - atoms with predicates from P. A goal is a conjunction of program atoms. The lack of a allowed constraint in the goal is no handicap. We can instead have an extra 0-adic atom A in the goal with a single rule A <- C : true.

A state of the computation is a pair <G,S,C> where G is a multisets of program atoms, S is a satisfiable multiset of allowed constraints, and C is a multiset of constraints. The computation rule selects some multisubset G'={B1,..,Bk} of atoms from G, and some multisubset of C' of the constraints in C. Let

A1 <- C1: G1, ... ,Ak <- Ck : Gk

be variants of k clauses for the predicates of the selected atoms. A next state of the computation is

<G-G'$\cup$G1$\cup$..$\cup$Gk, S',

C-C'$\cup$C1$\cup$..$\cup$Ck$\cup${B1=A1,..,Bk=Ak}>

where S' is $S \cup C'$ if $C \models (\mathbf{E})S,C'$, **false** if $C \models \sim(\mathbf{E})S,C'$. A computation terminates in success if G=**true**, C={} with the computed answer the subset of the satisfiable constraints S *related* to G. It terminates in failure, if S = **false**. The constraints related to G are those constraints that share a free variable with G or some other constraint related to G.

The following soundness and completeness results apply to any instance of this scheme (Maher 1987).

(R7.1.1) *Soundness*

If G has a computed answer C' then $C, P \models (\mathbf{V})[C' \rightarrow G]$

(R7.1.1) *Strong completeness*

If P, $C \models (\mathbf{V})[C \rightarrow G]$ for some constraint C then for any computation rule, G has a k successful derivations with final constraints C1,..,Ck such that $C \models (\mathbf{V})$ [C -> C1^$\vee$...$\vee$Ck^] where Ci^ is the existential quantification of Ci with respect to all variables not in C.

As an example of this result, Maher gives the example of the program

p(a,b)

p(X,b) <- X≠a:true

where C is HET. For the constraint Y=b and goal p(X,Y), we have

P,HET $\models$ (VX,Y)[Y=b -> p(X,Y)]

but we need both the computable constraint answers Y=b,X=a and Y=b,X≠a to cover the constraint Y=b. We have

HET $\models$ (VX,Y)[Y=b -> Y=b,X=a $\vee$ Y=b,X≠a]

When the constraints are limited to conjunctions of equations, then k =1 in the above result because of the strong compactness of sets of equations (Lassez et al 1988).

(R7.1.3) *Soundness and completeness of negation as failure*

For goal G, comp(P),$C \models \sim(\mathbf{E})G$ iff for a fair computation rule every branch of the computation tree for G is terminates

in failure.
This is the generalization of result (R3.3.5) for negation as failure. If C includes the normal axioms for equality, the stronger form of this result should hold (I have not checked the details):
(R7.1.4) For goal G, if every branch of the computation tree terminates and C1,....,Cn are the answers for the success branches, then comp(P),C |= (V)[G <-> C1^ V...V Cn^].

If the allowed constraints are closed under existential quantification and negation, we can use this result to allow negated atoms to return answers as in the SLDCNF scheme.

Maher (1986) extends the above scheme to incorporate the notion of committed choice with the concept of suspension until some subset GC of the allowed constraints of a clause is *valid* for the current environment of satisfied constraints S, or is the only satisfiable constraint of the alternative clauses. GC is *valid* if it can be satisfied for all values that satisfy S. The unifications with the input argument terms and the evalaution of the guard calls in Parlog and GHC meet this validity condition. The satisfiability condition is similar to the commit rule of P-Prolog.

Saraswat (1988) further refines this scheme to include an *ask* component which must be valid and a *tell* component which must be satisfied, atomically or eventually. In our presentation of the CLP scheme, all constraints are satisfied eventually.

Instances of the scheme

In any instance of the CLP scheme, checking whether or not some multiset of constraints is satisfiable for theory C must be implemented as an algorithm. We cannot have the unit of computation be an inference from some first order theory. As Jaffer and Lassez (1987b) remark, checking solvability should also be *incremental* - when the computation rule selects extra constraints C' to be checked with the existing solvable constraints S, it should not be necessary to recheck S. Also, solvable sets of constraints should have a *canonical form*, an equivalent simplified representation using a minimal number of constraints. This would be used for presenting answers and, ideally, it would also be used for the incremental checking of solvability. This may not always be possible, or the the minimal representation suitable for presenting answers may be different from that needed to check solvability. For SLD and AGLD, this canonical form is a substitution, for Prolog II it is a reduced set of equations and inequations. As with the E-unifiability scheme, the great strength of the CLP scheme is that it provides a logical framework for extensions and modifications of the unification based SLD. We simply need to ensure that the algorithms that replace unification when applied to some expression C correctly determines whether or not C |= (E)C for some consistent first order theory C of the constraint predicates. We then know that the above logical properties hold of the the computed answers. If the algorithm reduces C to a solution form S, we also need to establish that C |= (V)[C <-> S].

Prolog III (Colmerauer 1987) is an extension of Prolog II where the constraints are equations and inequations over terms, inequalities and linear equations of a special form over rational numbers, and boolean expressions over truth values.There is one non-free term constructor . for list concatenation enabling constraint equations such as X.Y.X= [1,2,3,4,1] to be used and solved. The constraint language is restricted to allow algorithmic reducibilty to solution form of any allowed constraint.

CLP(R) (Jaffer & Michaylov 1987) has equations over t(F,V), and inequalities and equations of arithmetic expressions over the real numbers. The implementation only checks the solvability of the term equations, arithmetic inequalities and linear equations. Non-linear equations are stored and checked only if the other constraints determine values for some of the variables that make them linear. If this does not happen, the non-linear equations remain as a qualification on the answer returned.

CIL (Mukai 1985), CS-Prolog (Kawamura et al 1987), CAL (Akiro et al 1988) and CHIP (Dincbas et al 1988) are other constraint languages.

8 Concluding remarks

What does the future hold regarding logic languages. I anticipate much activity in the area of algorithms for checking solvability of richer and richer and sets of constraints, extending the application of logic programming into new areas.

The committed choice languages will be further refined and will further converge to become powerful system building languages for multiprocessor machines.

Finally, I expect considerable impact from the recent developement of languages incorporating committed choice and parallelism and either or=parallel or or sequential search.

References

Akiro, A., Sakia, K., Sato, Y., Hawley, D., Hasegawa, R. (1988) Constraint logic programming language CAL, FGCS88, ICOT.

Apt., K.R., Blair, H., Walker, A., (1988), Towards a theory of declarative knowledge, in (Minker 1988).

Battani, G and Meloni, H. (1973) Interpreteur du Language de Programmation PROLOG, Groupe Intelligence Artificielle, Université Aix-Marseille II.

Bancilon, F., Ramakrishnan, R., An Amateur's introduction to recursive query processing strategies, ACM Int. Conf. on Management of Data, 1986.

Brand,P.,Haridi,S.,Warren,D.H.D. (1988) Andorra Prolog - the language and application in distributed simulation, FGCS88,ICOT

Burt, A., Ringwood, G.A., (1988), The binding conflict problem in concurrent logic languages, Research Report, Parlog Group, Department of Computing, Imperial College.

Chan, D., (1988), Constructive negation based on the completed data base, ICLP5, MIT Press.

Ciepielewski, A., Haridi, S., (1984), A formal model for OR-parallel execution of logic programs; IFIP 84, North-Holland.

Clark, K.L., (1978), Negation as failure, in Logic and Data Bases (eds. Gallaire, H. and Minker, J.), Plenum Press.

Clark, K.L., (1979), Predicate logic as a computational formulism, Research report, Logic Programming Group, Department of Computing, Imperial College.

Clark, K. L., Gregory, S., (1981), A relational language for parallel programming, ACM Conf. on Functional Languages and Computer Architecture.

Clark, K.L., Gregory, S., (1986), PARLOG: parallel programming in logic, ACM Toplas 8(1).

Clark, K. L., Gregory, S., (1987) Parlog amd Prolog United, ICLP4, MIT Press.

Clark, K.L., McCabe, F.G., The Control facilities of IC-Prolog, in Expert Systems in the micro-electronic age (ed. D. Michie), Edinburgh University Press.

Clark, K.L., McCabe, F.G., Gregory, S., (1982), IC-PROLOG language features in in (Clark and Tarnlund 1982)

Clark, K.L., Tarnlund, S-A., (1982), Logic Programming, Academic Press.

Colmerauer, A., (1982), Prolog and infinite trees, in (Clark and Tarnlund 1982)

Colmerauer, A., (1982b), Prolog II Reference manual, Groupe Intelligence Artificielle, Universite Aix-Marseille II.

Colmerauer, A., (1984), Equations and inequations on finite and infinite trees, FGCS84, ICOT

Colmerauer, A., (1986), Theoretical Model of Prolog II, in Logic Programming and its applications (ed. Caneghan, M. V. & Warren, D. H. D.), Ablex.

Colmerauer, A. (1987) Opening the Prolog III universe, Byte August 1987.

Conery, J.S., (1987), Parallel execution of Logic Programs, Kluwer Academic Publishers.

Degroot, D., (1984), Restricted and-parallelism, FGCS 84, ICOT

Dincbas,M.,van Hentenryck, P., Simonis,H., Aggoun,A., Graf,T., Berthier,F.(1988) The constraint logic programming language CHIP, FGCS88, ICOT.

Elcock, E. W., (1988), Absys: The First Logic Programming Language - A retrospective and a commentary, to appear in JLP.

van Emden, M.H., Lloyd, J.W., (1984), A logical reconstruction of Prolog II, ICLP3, Upsalla University.

Foster, E.M., Elcock, E.W., (1969), Absys 1: an incremental compiler for assertions, in Machine Intelligence 4 (ed. Michie, D.), Edinburgh University Press.

Hill, R., (1974), LUSH-resolution and its completeness, DCL Memo 78, Department of Artificial Intelligence, Edinburgh University.

Jaffar, J., Lassez, J-L., Lloyd, J.W., (1983), Completeness of Negation as failure rule, IJCAI-83.

Jaffer, J., Lassez, J-L., Maher, M.J., (1986), A Logic Programming Language Scheme, in Logic Programming Functions, Relations and Equations (ed. Degroot, D. & Lindstrom, G.). Prentice-Hall.

Jaffer, J., Michaylov, S. (1987) Methodology and implementation of a CLP system, ICLP4, MIT Press

Joxan, J., Lassez, J-L., (1987), Constraint Logic Programming, POPL, ACM.

Jaffer, J., Lassez, J-L., (1987b), From unification to constraints, in Logic Programming 87, LNCS 315, Springer-Verlag.

Kasif, S., Kohli, M., Minker, J., (1983), Prism: a parallel inference system for problem solving, IJCAI 83.

Kawamura, T., Ohwada, H., Mizoguchi, F., (1987), CS-Prolog: A generalised constraint solver, in Logic Programming 87, LNCS 315, Springer Verlag.

Kliger, S., Yardeni, E., Kahn, K., Shapiro, S., (1988), The Language FCP(!,:,?), FGCS 1988, ICOT

Kowalski, R. A., (1974), Predicate logic as a programming language, IFIP.

Kunen, K., (1987), Answer sets and negation as failure, ICLP-4, Melbourne, MIT Press.

Kunen, K., (1988), Some remarks on completed data bases, ICLP5, MIT Press.

Kunen, K., (1988b) Signed data dependencies in logic programs, to appear in JLP.

Lassez, J. L., Maher, M.J., Marriot, K.L., (1988), Unification revisited, in Foundations of deductive data bases and logic programming (ed. Minker, J.), Morgan Kaufmann.

Lloyd, J.W., Topor, R.W., (1986), A basis for deductive data base systems II, JLP 3(1).

Maher, M.J., (1986), Logic semantics of a class of committed choice programs, in ICLP 4, MIT Press.

Martelli, A., Montanari, U., (1982), An efficient unification algorithm, ACM Toplas, 4(2).

Miller, D., Nadathur, G. (1986) Higher order logic programming, ICLP3, LNCS 225, Springer-Verlag.

Minker, J., (1988), Foundations of deductive data bases and logic programming, Morgan Kaufmann.

Moto-Oka, T., Tanaka, H., Aida, H., Hirata, K., Maruyama, T., (1984), The architecture of a parallel inference machine - PIE, FGCS 84, North Holland.

Mukai, K., (1985), Unification over complex indeterminates in Prolog, TR-113, ICOT

Naish, L., (1984), Heterogeneous SLD Resolution, JLP 1(4).

Naish, L., (1985), Negation and Control in Prolog, LNCS 238, Springer-Verlag.

Naish, L., (1986), Negation and quantifiers in NU-Prolog, ICLP3, Springer-Verlag.

Naish, L (1988), Parallelizing NU-Prolog, ICLP5, MIT Press.

Pollard, S. H., Parallel execution of Horn clause programs, Ph.D., Thesis, Imperial College, London.

Ramakrishnan, R., (1988), Magic Templates: A spellbinding approach to logic programs, ICLP5, MIT Press, 1988.

Robinson, J.A., (1979), Logic: Form and function, Edinburgh University Press.

Saraswat, V. J., (1987), The concurrent logic programming language cp: definition and operational semantics, POPL, ACM.

Saraswat, V. J., (1988), A somewhat Logical Formulation of CLP Synchronisation Primitives, ICLP5, MIT Press.

Shapiro, E., (1986), Concurrent prolog, a progress report, IEEE Computer 19(8).

Shapiro, E. (1988) The family of concurrent logic programming languages, to appear in ACM Computing Surveys.

Shepherdson, J.C. (1984), Negation as failure: a comparison of Clark's completed data base and Reiters closed world assumption, JLP1(1).

Shepherdson, J.C. (1985), Negation as Failure II, JLP2(3).

Shepherdson, J.C. (1988), Negation in Logic Programming, in (Minker 1988).

Takeuchi, A., Furakawa, K. (1986) Parallel logic programming languages, ICLP3, LNCS 225, Springer-Verlag.

Takeuchi, A.,Takahashi,K.,Shimuzu,H.(1987) A description language with and/or parallelism for concurrency systems and its stream based realisation, ICOT TR-229.

Thorn, J.A., Zobel, J., (1987), NU-Prolog Reference Manual, Department of Computing Science, Melbourne University.

Ueda, K., (1985), Guarded Horn clauses, in Logic Programing, LNCS 221, Springer-Verlag.

Vasey, P., (1986) Qualified answers and their application to transformation, ICLP 3, LNCS 225, Springer-Verlag.

Warren, D.H.D. (1987), OR-Parallel execution models of Prolog, in Tapsoft 87, LNCS 250, Springer-Verlag.

Wise, M., (1986), Prolog multiprocessors, Prentice-Hall.

Wolfram, D.A., Maher, M.J., Lassez, J-L., (1984), A unified treatment of resolution strategies for logic programs, ICLP2, Upsalla University.

Yang, R., Aiso,H. (1986) P-Prolog: parallel language based on exclusive relation, ICLP3, Springer-Verlag.