UPMAIL Technical Report No. 59
March 22, 1990

Logic Programming Schemes and their Implementations

Keith L. Clark

Dept. of Computing
Imperial College, London
Email: klc@doc.ic.ac.uk

and

Computing Science Dept.
Uppsala University
Box 520, S-751 20 Uppsala, Sweden
Phone: +46 – 18 – 18 25 00

Abstract

This paper offers a tutorial but incomplete survey of a succession of proposed logic programming language schemes. All of the schemes surveyed can be considered variants or descendants of the original Kowalski (1974) scheme, now referred to as SLD. SLD is a resolution inference system (Robinson, 1965) for Horn clauses. Some of the descendants of SLD that we survey are also resolution systems, the others are not. Semantic properties of the schemes are discussed and links between the abstract schemes and their implementations as logic programming languages are explored. Familiarity with the general ideas of logic programming is assumed.

1 Introduction

It is now 16 years since Kowalski(1974) described his scheme for logic programming based on a resolution inference system (Robinson 1965) for Horn clauses (clauses with at most one positive literal), now referred to as the SLD (Selective Linear resolution for Definite clauses). Unification is at the heart of the scheme, acting as both data accessor and data constructor (see Kowalski 1974). Since then, many extensions and variations of the Kowalski scheme have been proposed, the early ones being the negation as failure extension of SLD (Clark 1978), now referred to as SLDNF, and Colmerauer's rational tree scheme (Colmerauer 1982), the first non-unification, hence non-resolution scheme. This paper is a tutorial introduction to a succession of logic language schemes all of which can be viewed as descendants of the Kowalski scheme. It starts with SLD and ends with the very general CLP (Constraint Logic Programming) scheme of Jaffar and Lassez (1987).

An essential aspect of the survey is the presentation of unification as equation solving for the Herbrand interpretation of the functors. This is an old idea that dates back to Herbrand(1930). Colmerauer (1982), Martelli and Montanari (1982), Goguen and Meseguer (1986) and Lassez et al (1988) have also identified algorithmic equation solving as a key idea of logic programming. By viewing unification as equation solving we can more easily relate the unification based schemes to their implementations - particularly the implementations allowing and-parallelism - and we can see more clearly how the generalisation of this process of equation solving for the Herbrand interpretation leads to the non-unification based descendants of SLD. For

This paper is a revision and substantial extension of (Clark 1988) example, if we change the interpretation for which we are solving the equations to one in which the functors are still free but with a that domain can contain infinite rational trees, and modify our equation solver accordingly, we get Colmerauer's (1982) scheme.

The semantic properties of the schemes are discussed using concepts from (Clark 1978,1979). This was the first formulation that focussed on the semantics of the answer substitutions returned by the Kowalski scheme, interpreting them as conjunctions of equations denoting relations over the domain of the Herbrand universe of terms. Queries denote relations and computed answers must denote subrelations of the query relation for all *allowed* models of the program. This contrasts with the original semantics of van Emden and Kowalski (1976), which focused on the semantics of program defined predicates. These were defined as sets of ground atomic formulae. The relation/sub-relation semantics allows us better to relate the Kowalski scheme to the other schemes. The semantic differences between the schemes hinges on what the allowed models of the program are, and on what predicates, with what meaning, can appear in the computed answers returned. In SLD any model is an allowed model and only the predicate =, denoting equality, can appear in computed answers. The predicates that can appear in answers we shall call the *reserved* predicates. These are predicates not defined by the program but given a fixed meaning by some pre-interpretation associated with the scheme. An allowed model is then some extension of the pre-interpretation which makes each program clause true.

Issues concerning implementation of the schemes are discussed and related to various proposed logic programming languages. As already pointed out by Elcock (1990), we shall see that Absys (Foster and Elcock 1969), one of the first declarative programming languages, can with some justice be considered the first logic programming language. This is because it can be viewed as a partial implementation of one of our schemes, called AGLD. We shall also see that some coroutining Prologs are really implementations of a significant variant of SLD, even though SLD allows for a coroutining evaluation. This is a property that was first realized by Naish (1984), who also proposed the SLD variant. In section 4, there is a particularly detailed discussion of parallel implementations of the AGLD scheme, since this is also the abstract scheme for the many proposed concurrent (and-parallel) logic programming languages.

Schemes not based on first order classical logic, such as the lambda-Prolog of (Miller and Nadathur 1986) are not covered, nor are schemes based on bottom up evaluation such as the magic template scheme of (Ramakrishnan 1988) and others surveyed in (Bancilhon & Ramakrishnan 1986).

1.1 Logical preliminaries

Syntax

We shall assume countably infinite disjoint alphabets P, R, F, V of *program predicates, reserved predicates, functors,* and *variables* respectively. Each predicate and functor has an associated non-negative integer arity. Functors of arity 0 we shall also call *constants*. *R* always includes the two 0 arity predicates *true*, *false* and the binary predicate =. For different schemes it may include other predicates.

A *term* is a variable, a constant or of the form f(t1,..,tk) where f ∈ *F* is a functor of arity k>0 and t1,..,tk are terms.

An *atomic formula* (or *atom*) is a 0 arity predicate or of the form p(t1,..,tk) where p ∈ *P*∪*R* is a predicate of arity k>0 and t1,..,tk are terms.

A *literal* is an atomic formula A or its negation ~A.

We shall also use the following logical connectives:

← (if conditional)
→ (then conditional)
↔ (biconditional)
v (disjunction)
, (conjunction)

As quantifiers we shall use ∃ (existential), ∀ (universal). We assume the usual definitions of (well formed) formula, free and bound variables, open and closed formulas. In addition, we shall consider a set of atoms to be a formula.

We shall say that a term or formula is *ground* if it contains no variables.

The *existential closure* ∃F of a formula F is (∃Y1,..,∃Yk)F where Y1,..,Yk are all the free variables appearing in F.

We shall use t[X1,..,Xk] to denote a term containing the variables X1,..,Xk and F[X1,..,Xk] to denote a formula containing X1,..,Xk as free variables.

When giving examples of terms and atoms we shall follow the Edinburgh Prolog convention and use names beginning with lower case letters for functors and predicates and names beginning with upper case letters for variables. The context will determine whether a lower case name is a functor or predicate, although we shall generally follow the normal logic convention and use f,g,h etc for functors and p,q,r etc for predicates.

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). For each equation X1=ti, ti is different from Xi. ti is called the *binding* for Xi.

Examples: A1={X=f(Y,Z),Y=U,Z=g(W),W=b}, A2={X=f(Y),Y=g(X,d)},
A3={f(U,g(b)), Y=U, Z=g(b),W=b}

Let A be an assignment. A variable X bound by an equation X=t[X1,..,Xk] in an assignment *immediately depends* on each variable Xi of its binding term.

Example: In A1, X immediately depends on both Y and Z, Y immediately depends on U, Z immediately depends on W.

Let the *depends* relation be the transitive closure of the *immediate depends* relation.

Example: In A1, X depends on Y,Z,U,W. In A2 X depends on Y,X

A *Herbrand assignment* is an assignment {X1=t1,..,Xn=tn} in which no bound variable Xi depends on itself. We shall usually use H, primed or subscripted, for Herbrand assignments.

Example: A1 is a Herbrand assignment, A2 is not.

A *substitution* (sometimes called an *indempotent substitution*) is an assignment {X1=t1,..,Xn=tn} in which no bound variable Xi appears in any of the binding terms t1,..,tn. We shall usually use S, primed or subscripted, for substitutions. A substitution is clearly also a Herbrand assignment.

Example: Only A3 is a substitution.

A *grounding* substitution binds each of its variables to a ground term.

Application of an assignment A to a term t, or formula F, which is denoted t.A, F.A, is the replacement of each free occurence of a variable V bound in A, by the term to which it is bound, to produce a new term or formula.

Examples:
h(X,W).A1 is h(f(Y,Z),b)
p(X,Y).A1 is p(f(Y,Z),U)

Interpretations

An *pre-interpretation* is a triple <***D***,***F***,***R***>. ***D*** is some non-empty set called the domain of the interpretation. ***F*** is a function mapping each k arity functor in *F* into a k-adic function from D^k to D. ***R*** is a function mapping each k arity predicate in *R* into a subset of D^k. ***R*** must always map *true* into **true**, *false* into **false** and = into the equality relation for ***D***.

An *interpretation* I is an extension <***D***,***F***,***R***,***P***> of a pre-interpretation <***D***,***F***,***R***> where ***P*** is a function mapping each k arity predicate in *P* into a subset of D^k.

What we have informally called *the* Herbrand interpretation of the functors is a ***D*** which is the set of all ground terms and an ***F*** which maps each k arity functor f into the function <t1,..,tk> → f(t1,..,tk). This, coupled with the fixed interpretation of =, *false* and *true,* we shall call *the Herbrand Interpretation* and denote it by HI.

A *Herbrand interpretation* is any interpretation which is an extension of HI.

Denotations

Let ***M*** be the function mapping ground terms to their denotations for interpretation <***D***,***F***,***R***,***P***> defined by:
M(c) = ***F***(c) for c a constant
M(f(t1,..,tk)) = ***F***(f)(***M***(t1),..,***M***(tk))

A ground atom p(t1,..tk) is **true** for an interpretation iff
<***M***(t1),..,***M***(tk)> ∈ ***P***(p) when p ∈ *P*
<***M***(t1),..,***M***(tk)> ∈ ***R***(p) when p ∈ *R*.

An existential closure of a set of atoms C is true for an interpretation iff there is some allocation X1=d1,..,Xk=dk of k (not necessarily distinct) individuals d1,..,dk from ***D*** as respective denotations of X1,..,Xk such that each atom in C is then **true**.

Let T be a tuple of variables <X1,..,Xk> and let Y1,..,Ym be all the variables of a set of atoms C not contained in T.

The k-ary *relation* R^T_C *denoted* by C for an interpretation I is the set {<d1,..,dk>: for denotations X1=d1,..,Xk=dk, (∃Y1,..,Ym)C is **true**}.

Two sets of atoms C1,C2 are I-*equivalent*
iff for interpretation I, $R^T_{C1}=R^T_{C2}$, T a tuple of all the variables appearing in C1, C2.

Two sets of atoms C1,C2 are *equivalent*
iff they are I-equivalent for all interpretations I.

Two sets of atoms C1,C2 are *Herbrand equivalent*
iff they are I-equivalent for all Herbrand interpretations I

Note that these definitions also define equivalence of assignments. We assume the normal extension of these definitions to give denotations of closed or ground formulas (which are truth values) and to formulas with free variables (which are k-ary relations, k being the number of free variables).

A *model* of a set of closed well formed formulas is any interpretation for which each formula denotes **true**.

We shall use $\models$ for logical implication. Let F1 and F2 be formulas.

F1 $\models$ F2 iff for every interpretation the denotation of F1 is a subset of the denotation of F2.

false is a subset of **true**. We shall use **false** to denote the empty set extension for a relation.

We can now identify a key semantic difference between assignments and substitutions and relate substitutions to herbrand assignments.

R1.1

The existential closure of a substitution S is true for every interpretation, or, equivalently, the relation $R^T{}_S \neq$ **false** for any interpretation, T a tuple of all the variables in S

Proof

We can assign any elements of ***D*** to the variables {Y1,..,Yn} of the substitution {X1=t1,..,Xk=tk} that appear in the terms t1,..,tk. By definition, no Xi is included in {Y1,..,Yn}. We then assign to X1,..,Xk the resulting denotations of t1,..,tk.

R1.2

For every Herbrand assignment H there is an equivalent substitution S.

Proof

Let n be the maximum length of any chain of variables Y1,..,Yj in H such that Yi immediately depends on Yi+1 for i=1,j-1. n must be finite since there are a finite number of variables in H and by definition no variable in H depends on itself. So there can be no infinite chains. Proof is by induction on n.

If n=1, H is already a substitution.

If n>1. Assume true for any H with maximum length chain n-1. Let W1,..,Wk be all the variables bound by H which are beginning points of chains of length n. Replace the binding equation Wi=ti for each such variable by the equation Wi=ti.H. The new assignment H' is equivalent to H. H' has n-1 has its maximum length chain so it can be transformed into an equivalent substitution S.

Example: The above proof gives us the following step by step transformation of A1 to A3.

{X=f(Y,Z),Y=U,Z=g(W),W=b}
=> {X=f(U,g(W)),Y=U,Z=g(W),W=b}
=> {X=f(a,g(b)),Y=U,Z=g(b),W=b}

R1.3

Every Herbrand assignment denotes a non-empty relation for every interpretation.

Proof

Follows from properties 1 and 2. The existential closure of a non-Herbrand assignment does not have this property. For example, A2 does not denote a non-empty relation for any Herbrand interpretation. It only denotes non-empty relations for interpretations in which X=f(g(X,d)) has a solution.

2 Unification as equation solving and special purpose inference

Unification as equation solving for the Herbrand interpretation

The essential computational step of SLD as traditionally described is the unification of one or more pairs of terms {(t1,t'1),...(tk,t'k)}. This is the computation of a *most general unifier* which is a substitution S such that for each pair (ti,t'i) ti.S is syntactically identical to t'i.S. We refer the reader to Robinson(1979) for the definition of a most general unifying substitution. For a thorough treatment of many of the issues concerning unifying substitutions see also (Lassez et al 1988).

Following (Martelli and Montanari 1982) and (Herbrand 1930) we can recast the problem of finding a most general unifier of {(t1,t'1),...(tk,t'k)} as the problem finding a *solution* of the set of equations

E={t1=t'1,t2=t'2,...,tk=t'k}

for the Herbrand interpretation HI that can be expressed as an *equivalent* Herbrand assignment

H≐{X1=t"1,...,Xn=t"n}

Equation solving algorithm

Algorithm 1

The following algorithm, which is a modification of that given by (Martelli and Montanari 1982), and originally (Herbrand 1930), reduces a set of equations E={t1=t'1,..,tk=t'k} to a set of atoms which is Herbrand equivalent to E in the context of a Herbrand assignment H. The algorithm terminates with a Herbrand assignment H'=H∪H" (a success termination) or it terminates with a set of equations augmented with the atom *false*, (a fail termination).

Repeat the following step until E is {}, or *false* ∈ H. On termination return H as the answer.
Select any equation e in E and *delete* it from E.
Cases:
(a) e of the form f(t1,..,tk)=f(t'1,..,t'k). Add the k equations t1=t'1,..,tk=t'k to E.
(b) e of the form X=X. Do nothing.
(c) e of the form X=t, or of the form t=X, where t is not X.
  Subcase (i) There is an equation X=t' in H. Add t'=t to E.
  Subcase (ii) No X=t' equation in H and H∪{X=t} is a Herbrand assignment, add X=t to H
  Subcase (ii) No X=t' equation in H and in H∪{X=t} X depends on itself (the occur check), add *false* to H
(d) e of the form f(...)=g(...), where f and g are different. Add *false* to H.

The above algorithm always terminates. Notice that each step of the algorithm is such that it transforms a set of equations into a Herbrand equivalent set of equations. Note also that H∪{X=t} is a Herbrand assignment iff the occur check for X fails, so there is really only one test to perform before X=t or *false* is added to H.

If it terminates with *false* added to H, the original set of equations H∪E has no solution for HI, i.e. it is Herbrand equivalent to *false*.

If it terminates with a Herbrand assignment H' then the equivalent substitution S', generated by the proof of proposition 2, is a most general unifier of E.S, S the substitution equivalent of H. The proof is a simple extension of that given in (Martelli and Montanari 1982) for their algorithm. *However, the really important property of the algorithm is that it generates an H' that is Herbrand equivalent to the initial H∪E.*

The algorithm corresponds fairly well to what happens in many implementations of logic programming languages. These also represent the result of a unification as an assignment rather than as a substitution. As an evaluation proceeds, unification of a new call is performed relative to the incrementally generated assignment representing the results of unifying all its ancestor calls.

Algorithm 2

We can parallelise the algorithm, providing we impose a restriction. We can allow a subset E' of equations from E to be selected and deleted from E at each step. We then simultaneously apply the rules to each equation e in E', subject to the restriction that we apply rule (c)(ii) to at most one equation X=t for each variable X. Any other equation X=t' in E' is a simply added back to E. Thus, at each step we add at most one binding for each variable to H'. This restriction corresponds to a multiprocessor implementation in which a variable binding is held in a single memory location and assignment to this location is an atomic operation.

Algorithm 3

An alternative parallelisation splits the initial set E into k+1 subsets E0,E1,..,Ek with E0 initially {} and the algorithm is applied to H,E0,..,H,Ek independently. However, instead of each new binding generated by the reduction of Ei being transfered to H, it is instead transfered to a local Hi which is initialized to {}. Rule (c) is modified so that both H and Hi are checked for previous bindings for a variable. The modified rule is:

(c') ei from Ei of the form X=t, or of the form t=X, where t is not X.
Subcase (i) There is an equation X=t' in $H \cup Hi$. Add t'=t to Ei.
Subcase (ii) No X=t' equation in $H \cup Hi$ and $H \cup Hi \cup \{X=t\}$ is a Herbrand assignment, add X=t to Hi
Subcase (iii) No X=t' equation in $H \cup Hi$ and in $H \cup Hi \cup \{X=t\}$ X depends on itself (the occur check), add *false* to H.

Rules (a),(b),(d) remain the same. In addition, as soon as some Ei has been reduced to {} its corresponding Hi is added to E0. When all E0,..,Ek have been reduced to {}, $H'=H \cup H0$ is the output of the algorithm.

This corresponds to a multiprocessor implementation in which, say, several different calls are initially independently unified in parallel with heads of corresponding clauses and the resulting bindings are incrementally reconciled (by being added to E0) as each individual unification terminates.

Adding an equation X=t to an Hi amounts to *broadcasting* the binding to all the other equations in Ei. It is equivalent to substituting for X in all the other equations because, unless the algorithm previously terminates with failure, each other equation containing X in Ei will eventually be reduced to an equation to which case (c)(i) applies. Thus the algorithm allows only local broadcasting of bindings generated from each Ei to other equations descended from Ei until Ei={}. The adding of Hi to E0 is the attempt to reconcile these locally generated bindings with those generated by each other unification process. However, the reconcilation with the bindings Hj generated from Ej begins only when Ej has been reduced to {}.

Algorithm 4

We start the algorithm with some global Herbrand assignment H and split E into E1,..,Ek with their local H1,..,Hk which are each initialized to {}.

We then in parallel apply the rules of algorithm 3 to each H,Hi,Ei. The difference is what happens when some Ei={} and its Hi is a Herbrand assignment. The following rules can be applied in parallel to any H,Hi,Ei for which this is the case.

(e) H contains no binding for a variable bound in Hi and $H \cup Hi$ is a Herbrand assignment. Transfer all the bindings from Hi to H atomically, i.e. without allowing any other application of rule (e) to add a binding for a variable bound in Hi to H.
(f) H contains no binding for a variable bound in H but Hi contains a binding for a variable X which depends on itself in $H \cup Hi$. Add *false* to H.
(g) H contains bindings Y1=t'1,..,Ym=t'm for variables Y1,..,Ym bound in Hi by

equations Y1=t1,..,Yj=tm. (These will have been added to H by rule (e) applied to some other H,Hj after the bindings Y1,..,Ym were added to Hi.) Set Ei={t'1=t1,..,t'm=tm} and delete Y1=t1,..,Yj=tm from Hi.

This algorithm corresponds to an implementation in which several calls can by concurrently unified in a quasi independent way. However, application of rule (e) allows early communication of the results of any unification which terminates providing all its generated bindings are compatible with the current global set of bindings H. The requirement that all of Hi is transferred atomically to H is the concept of *atomic* unification for a scheme or a language that allows parallel evaluation of calls. We shall return to this idea in section 4.

A further refinement would effectively anticipate application of rule (g) whenever bindings are added to H by rule (e). When this happens, rule (g) can immediately be applied to each of the other unification processes which have not yet terminated, i.e. to which rule (e) has not yet been applied.

Unification as specialised inference from a theory of the Herbrand Interpretation

When we consider semantic properties of schemes, because they are logic programming schemes, we will want to be able to characterize the answers computed as logical consequences of some theory associated with the program. To this end, we shall find it useful to interpret unification as an inference. Fortunately, we can do this. We can axiomatise the Herbrand interpretation of *F* and = in such a way that equation solving unification becomes just specialised inference from these axioms.

The following *freeness* axioms, FR, from (Clark 1978), characterize the Herbrand interpretation of *F*.

(F1) for every functor $f \in F$
  $f(X1,..,Xk) = f(Y1,..,Yk) \rightarrow X1=Y1,..,Xk=Yk$
(F2) for every pair of distinct functors $f,g \in F$
  $f(X1,..,Xk) \neq g(Y1,..,Yn)$
(F3) for every non variable term t[X] containing some variable X, $X \neq t[X]$

The following equality axioms, EQ, characterise the equality interpretation of =

$X=X$,
$X=Y \leftarrow Y=X$,
$X=Z \leftarrow X=Y, Y=X$
$f(X1,..,Xk) = f(Y1,..,Yk) \leftarrow X1=Y1,..,Xk=Yk$ for each $f \in F$,
$p(X1,..,Xk) \leftarrow p(Y1,..,Yk), X1=Y1,..,Xk=Yk$ for each $p \in P \cup R$ other than =

EQ+FR comprise the Herbrand equality theory HET characterizing the Herbrand interpretation HI.

Each step of each 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$\models$ $R^T{}_{H\cup E}=R^T{}_{H'}$, T a tuple of all the variables in E

$H'=H\cup H''$ is the output of the unification process. That $R^T{}_{H\cup E}\supseteq R^T{}_{H'}$ only needs the general equality axioms EQ. That $R^T{}_{H\cup E}\subseteq R^T{}_{H'}$ needs the freeness axioms FR.

As corollaries of R2.1 we have:

R2.2 *false* $\in$ H' iff HET$\models$ $R^T{}_{H\cup E}=$**false**
R2.3 H' is a Herbrand assignment iff $R^T{}_{H\cup E}\neq$**false** for any I

Key properties of the equation rewrite algorithm

Unification has three very desirable properties:
(1) A simple test on the syntactic form of the output $H'=H\cup H''$ (whether or not it contains *false*) tells us whether or not E has a solution for theory HET in the context H.
(2) In the case that it has a solution H'' is a more explicit representation of the set of all solutions than E.
(3) It is incremental, if we want to check whether or not some set $E'\cup E$ has a solution for HET in context H, and we have already reduced $H\cup E'$ to an equivalent Herbrand assignment H', we can simply apply the algorithm to E in context H'.

We shall say that H produced by the equation rewrite algorithm is an *incremental solution form* for any set of equations E for the theory HET.

Generalizing the equation rewrite

If we replace our equation rewrite algorithm by some other operation O, it is desirable to retain all three properties of the algorithm.

To ensure that the new scheme is a logic programming scheme, we *must* require that O can be interpreted as checking whether or not some set of equations has a solution for some theory T. More generally, O can be checking that some set of formulas C of restricted form (usually atoms), containing only predicates from the reserved set *R*, has a solution for the theory T. T describes the pre-interpretation of *F* and *R* associated with the scheme. We shall call T the *reference theory* for O.

In checking this property, O might generate a normalised form N for C. Minimally, N must be T equivalent to C. In the worst case N could be just C. Ideally, N is in some sense simpler or more explicit than C.

Finally, when O does generate a simpler normalised form N, it is desirable that O is incremental. As with our HI equation solving, we should be able to check some set $C'\cup C$ for which C' has a normalised form N' by applying a O to $N'\cup C$.

(Paterson & Staples 1988) present a generalised treatment of unification and constraint solving algorithms.

Schemes with incremental solution forms

As we shall see in detail later, (Colmerauer 1982) is an example of a logic programming scheme in which equations are checked for solutions for an equality theory of infinite rational trees. This scheme, which we shall call RTS, solves the equations for this theory by an equation rewrite algorithm that generates an incremental solution form which is an assignment A.

The (Colmerauer 1984) scheme generalises RTS in allowing both equations and inequations in the set of atoms to be checked for solution. The reference theory is an extension of that for RTS, as is the equation/inequation solving algorithm. The algorithm also generates an incremental solution form.

The minimum requirement

As we have already indicated, the minimum requirement is that O checks for solvability of some set of formulas C containing only predicates from the reserved set *R* for some reference theory T. Without this, we will not be able to claim that answers computed by the scheme are logical consequences of some set of first order sentences. In the worst case, O can be the concurrent attempt to prove both ∃C and ~∃C from the axioms of the theory T. For this to be guaranteed to terminate, T must be what Jaffar and Lassez(1987) call *satisfaction complete.* It must be the case that for all allowed sets of formulas C, either ∃C or ~∃C is provable from T. This is the minimal requirement of their CLP scheme.

As an example, HET is satisfaction complete for sets of equations. In fact, it is satisfaction complete for any set of first order formulas C using only the predicate =. This result is given in (Kunen 1987).

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.

Syntax

For SLD, the set of reserved predicates *R*={=, *false*, *true*}.

An SLD program is a set of definite clauses P.

A *definite clause* is an implication of the form

r(t1,..,tn) ← A1,..,Ak, k ≥0

where A1,..,Ak are atoms with predicates from *R*∪*P*. The clause is implicitly universally quantified with respect to all its variables. The clause is *about* the predicate

$r \in P$. r(t1,..,tn) is the *head* of the clause, A1,..,Ak is the body. The body atoms we shall also refer to as *calls*.

Operational semantics

A computation is invoked by a *query* or *goal* which is a multiset of calls G written as a conjunction B1,..,Bm.

A *state* of the computation is a pair <G,H> where G is a multiset of calls and H is a Herbrand assignment which may contain *false*.

The initial state is <{B1,..,Bm},{}>.

A *success* termination state is any state of the form <{},H>, *false* $\notin$ H.

A *fail* termination state is any state of the form <G,H$\cup$ *false* >.

The computation is a non-deterministic evaluation branching from the initial state.

The computation is controlled by a *computation rule* CR. This is a rule, which for any non-terminal state <G,H> selects exactly one call B=r(t'1,..,t'k) from the multiset of calls G.

The number of next states of the computation is the number of clauses in P about the predicate r of the selected call.

Let

r(t1,..,tn) $\leftarrow$ A1,..,An

be a clause in P about r with its variables renamed so that its set of variables is disjoint from the set of variables of the state <G,H>.

A next state is <G-{B}$\cup${A1,..,An},H'> where H' is the result of applying equation solving algorithm 1 to H and E={t1=t'1,..,tk=t'k}.

The *search tree* for a query G is the finitely branching tree routed at <G,{}>. The offsprings of a node in the tree are all the j different states that can be generated using the j clauses for the predicate of the call selected in that state. The strategy for constructing the search tree is the *search* strategy. The strategy is *fair* if it does not indefinitely postpone the construction of some branch of the search tree.

Answers to the query G are given by success terminating branches. The answer computed by a success branch terminating in <{},H> is the substitution S which equivalent to H (R1.2), restricted to bindings for the tuple of variables T=<X1,..,Xk> of G. If G contains no variables, the answer is **true**.

Logical semantics

The following logical properties of SLD, strengthening the soundness and completeness results of (Hill 1974), were proved in (Clark 1979).

R3.1.1 *Soundness*

For every CR-computed answer substitution S for query Q and program P,

$P \models R^T{}_S \subseteq R^T{}_G$

Since for a substitution, $R^T{}_S \neq$ **false** for any interpretation I (R1.1), this result implies the usual but weaker soundness result :

If there is a success computation path then $P \models (\exists)G$

R3.1.2 *Independence of the computation rule*
Let CR, CR' be two computation rules. For each CR-computed answer S there is an CR'-computed answer S' which is equivalent to S.

R3.1.3 *Strong completeness*
For every substitution S' which binds the variables T such that $P \models R^T{}_{S'} \subseteq R^T{}_G$ there is an CR-computed answer S such that $R^T{}_S \supseteq R^T{}_{S'}$ for every interpretation I.

The proof of R3.1.3 in (Clark 1979) uses the fact that algorithm 1 returns an H for which the equivalent substitution is a most general unifier of the selected call and the head of the program clause, but I am sure it can be proved using only R2.1, the key semantic property of the algorithm.

Differences between the search trees for different computation rules
It follows from R3.1.2 that the number of success terminating branches on a search tree is independent of the computation rule. In fact, a trivial consequence of the results in (Clark 1979) (but proved in (Lloyd 1984)) is that the length of the success branch computing a given answer is the same on each computation tree. As pointed out in Chapter 5 of (Naish 1985), the search trees for different computation rules differ, if at all, only with respect to the length and number of the failing or infinite branches. As Naish concludes, the crucial role of the computation rule is "to avoid unnecessary failure (and infinite) branches, as much as possible".

Possibilities for parallel evaluation

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,H> that select atoms which can be independently unified with heads of clauses in the context H. 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, where S is the substitution which is equivalent to H. This is *independent and-parallelism.* On the issue of efficiently testing for independence see (Degroot 1984) and (Hermenegildo & Rossi 1989).

Qualified answers

Another simple extension, is to allow the return of *qualified answers.* Suppose the original goal has been reduced to a state <G',H'> where H' is a Herbrand assignment but $G' \neq \{\}$. Let S be the subset of bindings of the substitution S' equivalent to H' that only binds variables in query G. A qualified answer (Vasey 1986) for that computation path is S, G'.S' . A simple generalization of R3.1.1 tells us that

$$P \models R^T{}_{S,G'.S} \subseteq R^T{}_G$$

Of course, $R^T{}_{S,G'.S'} \neq$ **false** only if $P \models \exists(G'.S')$

Types of computation rule

Let us suppose that the calls in the G component of the state are double indexed. The first index is the computation step at which they are introduced, the second is the position of the call in the original query or the clause body that introduced the call. So the i'th call of the original query goal has index 0,i. The j'th call of the body of the clause used at the k'th step down some computation path has index k,j. This indexing corresponds to a stack implementation.

A rule that always selects a call with a highest first index is a *depth first* rule. The rule that always selects the call with the lowest second index amongst those with a highest first index is the *leftmost call* rule. A rule that does not always select a call with a highest first index 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 given goal and in the body of the program clauses.

Implementations of SLD

The first implementation was Prolog (Battani 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 search strategy trying the clauses in the fixed order in which they are entered with backtracking on reaching a fail termination. This is, of course, a non-fair search strategy; but it does allow efficient implementation.

A simple stack can be used to keep track of the state of the computation on the current branch of the search tree. The i'th entry in the stack is a pointer to the clause C used at the i'th step paired with an index <j,k>, j<i, which identifies the call with which the head of C was unified. The current backtrack point BP for the stack, which is separetely recorded, is the position of the most recent entry on the stack which points to a clause which is not the last clause for its call. Usually this is the top of the stack, but it need not be. When BP changes, its old value is remembered. When a selected call fails to unify with the heads of any of its clauses, the stack is popped to entry BP-1, the old value for BP is restored, and the next clause for the call of the popped PB entry is tried.

The great advantage of backtracking search is that at any one time it needs to remember only the bindings for variables of a single branch of the search tree. Variables can therefore be represented by pointers to a *single* memory location. When a binding equation X=t for a variable X is added to H, the value t is stored in the location assigned to X. If X is a variable of the query, or of some clause used at a computation step i where i≤BP, X is trailed, i.e. the fact that it has been bound is remembered. On backtracking to BP, the t binding for X is undone; X is reset to undefined. For an introduction to Prolog implementation techniques see (Maier & Warren 1988).

Implementations with more complex computation rules

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 G component of the state with different indices for different modes. A mode of use is an input/output pattern for the argument positions of the call. Suppose that we successfully unify a call r(t'1,..,t'k) in in state <G,H> with a head r(t1,..,tk). If i is an input argument position, the input restriction is satisfied if all the bindings that result from rewriting the equation t'i=ti are for variables in the clause. If j is an output argument position, the output restriction is satisfied if t'j is a variable that is not bound in H.

Implementations of courouting rules

A coroutining rule in IC-Prolog is specified by *data flow* annotations on variables in calls. To understand the semantics of the annotations we need to assume a more complex indexing of calls in G. The calls are indexed by tuples of positive integers but of arbitrary length. All the calls in the initial state have a single integer index giving the position in the query conjunction. Then, when a call with index <i2,..,ik> is selected, the i'th introduced call from the body of the used clause is given index <i,i2,..,ik>. With this indexing, the leftmost call rule is the rule that always selects an atom with a maximal length index with the lowest first index. A depth first evaluation of a call A is a computation that always selects a call with an index that is an extension of the index of A, if there is such a call. An IC-Prolog computation always starts using the leftmost call rule and, unless forced by a data flow annotation to select some other call, selects using this rule.

In IC-Prolog, a ? annotation on a variable V in a call B makes B an *eager consumer* of the binding for V. Suppose that before B is selected the computation rule selects a call A and the unification with the head of the next clause for A results in the generation of a binding equation V=t, t a non-variable. The body atoms for the clause are introduced into the goal with their normal index but the computation rule now selects B. It continues with a depth first evaluation of B until there are no more descendants of B, or until some descendant B' of B demands a further instantiation of the binding for V. In either case the computation rule switches back to select the call A' it would normally have selected after call A, had there not been a coroutining transfer to B.

B' demands a further instantiation of the binding for V if the unification of B' with the head of a clause C generates a binding equation V'=t' where t' is a non-variable and V appears in t. This binding is not added to H, indeed none of the bindings generated by the unification of B' are added to H. When the computation rule swtiches back to A' it is as though B' had never been selected.

The eager consumer role of B for the binding of V is inherited by the descendants of B. So, again, if some other call A" is selected before B' is selected, and the unification of A" with some clause head adds a binding equation V"=t" to H, where t" is a non-variable and V" is a variable in t, the computation rule switches back to the depth first evaluation of B, restarting with B'. There is a renewed attempt to unify B' with the head of the clause C the attempted use of which caused the earlier switch out of the depthfirst evalaution of B. Notice that C may not be the first clause for the predicate of B'. When B' had been previously selected, before the switch back to A', the backtracking search might have discovered, *without* generating a demand for a further instantiation of V, that all the clauses before C for call B' all lead to a fail state. In strict accordance with the SLD scheme, when B' is now selected, the fact that it was previously selected and then deselected should not be remembered. The backtracking search should start by trying to use the first clause for B'.

The Naish HSLD scheme, that we shall describe in the next section, addresses this point. In the Naish scheme, a resumption of the depthfirst evaluation of B could correctly start by trying to reuse clause C for B'. The results of the earlier failed attempts to evaluate B' are explicitly remembered in the scheme, as they are in the IC-Prolog implementation. So IC-Prolog is really an implementation of HSLD.

The eager consumer annotation allows early incremental checking of the incrementally generated binding for a variable V. It can result in *earlier failure*, a rejection of the binding for V before it has been completed constructed. (See the remark above about search tree differences for different computation rules.)

A dual notion, that of a *lazy producer*, is specified by the annotation ^ on the variable V in a call B. There is a switch *to* B if a call which is selected before B tries to bind V to a non-variable and the depthfirst evaluation of B then lazily generates the binding. Strict one step alternation between the depth first evaluation of two or more calls can also be specified, allowing 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 substitution equivalent of the current set of bindings H, then evaluation continues as though the call was B. If V is unbound, B is temporarily removed from the goal component 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 generated 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 now added to the goal component together with A1,..,An. B and the other suspended calls are given new indices. These are the indices that they would have had if they had preceded A1,..,An in the used clause. The freeze condition is not inherited, a descendant of B will resuspend only if it is contained inside another explicit freeze call. Note that when a frozen call in Prolog II is reintroduced, its backtracking evaluation always starts with the first clause for its predicate because no attempts have been previously made to evaluate the call. So use of the freeze call is in accordance with the coroutining allowed by the SLD scheme.

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 component of the state.

In MU-Prolog (Naish 1985) coroutining is also implemented by temporarily removing a call from the goal component. Here, the suspension condition is not specified by the form of call, but by the specification of allowed modes of use of the clauses for its predicate. Optionally, for a predicate r a set of modes of use can be 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. Multiple wait statements are used when there are several possible modes of use for the same set of clauses.

When the currently selected call B is for a predicate with *wait* statements, the call is suspended if the unification with the next clause C for r would result in the generation of non-variable binding for variables V1,..,Vm which are disallowed by some by *wait* statement for r . The call B is removed from the goal component and linked with each of V1,..,Vm. It is reintroduced with an index that makes it the next selected call as soon as a non-variable binding is broadcast for any of the variables V1,..,Vm and the unification with the head of clause C is retried. So, like IC-Prolog, this is really an implementation of the HSLD scheme.

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 depth first evaluation would result in an infinite branch in the search tree.

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 used. 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 of several variables is bound to a non-variable term.

Parallel implementations

(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 of a clause depending upon which variables in the clause are bound by the head unification. Warren (1987) is a survey of recent work on the or-parallel implementations of Prolog.

3.2 Heterogeneous SLD - the Naish scheme

An interesting variation of the SLD, called Heterogeneous SLD (HSLD), was proposed by Naish(1984) as the correct abstract model for coroutining backtracking languages such as IC-Prolog and MU-Prolog. In these languages, the computation rule may return to a previously selected call B, which was deselected because it caused a corouting jump, and may restart the evaluation of B using other than the first clause for its predicate. This is because before B was deselected, all attempts to use the earlier clauses for B had resulted in failure. Unlike SLD, HSLD records the fact that the earlier clauses have been tried in the computation state.

Syntax

Programs and queries are as in SLD.

Operational semantics

In a state of the computation <G,H>, each call in G is explicitly linked with a set of the clauses for its predicate that can be used in generating a successor state. In the initial state, each call of the query is linked with the set of all the clauses for its predicate.

The HSLD computation rule, when applied to a state <G,H> returns a sequence Q of pairs <B1,C1>,...,<Bj,Cj>,...<Bn,Cn> where Bj is a call in G and Cj is a clause in the set of clauses linked with call Bj in G.

Let <Bj,Cj> be in the sequence Q returned for state <G,H>. Let new state <Gj,Hj> be generated from <G,H> as in SLD using Bj as selected call and Cj as the clause. All the introduced calls from the body of Cj are linked with a set of all the clauses for their predicates. The calls in Gj that descend from G inherit their sets of linked clauses except that in some cases clauses are deleted from the set. Suppose <Bi,Ci>, i<j, is an earlier pair in the sequence Q. Then Ci is deleted from the set of clause linked with Bi in <Gj,Hj>. However, Cj will appear in the set of clauses linked with the occurrence of Bj in <Gi,Hi>, the successor of <G,H> generated using Ci on call Bi. So the order of pairs on the sequence Q is important even though it is not intended to constrain the order in which successor states are generated by some search strategy.

Success and fail termination states are as in SLD.

Logical semantics

Naish proves that providing the sequence returned by the computation rule includes all the clauses still linked with at least one call B in the state <G,H>, then the scheme computes the same set of answers as SLD.

Implementations

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,H> ends in failure, to backtrack to <G,H>, select another call Bj in G, and subsequently to try to solve Bi in a successor state of <G,H> without needing to reconstruct the failure subtree generated by using C to try to solve Bi. This is because in HSLD, C can be deleted from the clause set associated with Bi in the new immediate descendant of <G,H>.

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 current backtrack state, retry the failed call if it is in the goal component of the state rather than the previously selected call. Repeat this until the call succeeds or until it is no longer in the current backtrack state.

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.

Syntax

An SLDNF program is a set of clauses written as implications of the form

r(t1,..,tn) $\leftarrow$ L1,..,Lk, k $\geq$0

where L1,..,Lk are literals. The implication is implicitly universally quantified with respect to all its variables. The clause is *about* the predicate r. The negated atoms in the body we shall call *negated calls.*

Operational semantics

An SLDNF computation is invoked by a query which is a multiset of literals.

States of the computation are as in SLD except that the goal component is a multiset of literals. The initial state is <G,{}> where G is the multiset of literals of the query. The success and fail termination states are as in SLD.

The computation rule selects exactly one call from goal component. However, the computation rule can select a negated call ~B in a state <G,H> only if B.S is ground, where S is the substitution that is equivalent to H. Following (Lloyd & Topor 1986), we shall call a computation rule that obeys this constraint on the selection of negated calls a *safe* rule.

Given a selected call L from the state <G,H> the possible next states are:

If L is an unnegated call B, the next states are generated as in SLD. There are k possible next states, k being the number of program clauses about the predicate of B.

If the selected call is a negative call ~B, then an auxiliary query evaluation is started with initial state <B,H>. If every computation path for <B,H> ends in failure, ~B is assumed to succeed. The next state of the main computation is <G-{~B},H>. If some computation branch ends in success, *false* is added to H giving a failure state <G,H$\cup${*false*}> as the next state of the main computation. This is the *negation as failure* rule.

A computation branch *flounders* if a state <{~A1,..,~An},H> is generated which has only negated calls and each call ~Ai is such that Ai.S is not ground.

As in SLD, a computed answer is given by each success termination state <{},H>. It is the substitution S which is equivalent to H restricted to the variables of the query.

Implementations

Prolog is an SLDNF system 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 ~B generates a binding for a variable in B.S.

Logical semantics

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 of the functors. Secondly, there is an implicit assumption that the given clauses somehow constitute a complete definition for each relation. To give a logical semantics we must assume that the given set of clauses is shorthand for some other, *complete* program.

Program completion

An SLDNF program clause has the form

(a) r(t1,..,tn) ← L1,..,Lk

The *homogeneous form* of the clause is

(b) r(X1,..,Xn) ← X1=t1,..,Xn=tn,L1,..,Lk

where X1,..,Xk are distinct variables not in (a)

The *general form* of the clause is

(c) r(X1,..,Xn) ← (∃Y1,..,Yj) (X1=t1,..,Xn=tn,L1,..,Lk)

where Y1,..,Yj are all the variables of (a).

Remember that unification between a selected call r(t'1,..,t'n) of state <G,H> and the clause head r(t1,..,tn) is the application of algorithm 1 to H,E with E={t'1=t1,...,t'n=tn}. This is equivalent to the application of the algorithm to H,E with E={X1=t'1,..,Xn=t'n}∪{X1=t1,..,Xn=tn}. If we revised our definition of a computation step, so that all n initial equations of the body of the homogeneous form of the clause are added to the equations {X1=t'1,..,Xn=t'n} before the algorithm is applied, computation using the homogeneous form of each 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) $\leftrightarrow$ E1 $\vee$ E2 $\vee$ ..$\vee$ Em.
This is implicitly universally quantified for all the variables X1,..,Xk.

Example
The completed definitions corresponding to clauses
on(U,[U])
on(U,[V|Z]) $\leftarrow$ on(U,Z)
and
female(X) $\leftarrow$ person(X),~male(X)
male(tom)
male(bill)
are
on(X1,X2) $\leftrightarrow$ ($\exists$U) (X1=U,X2=[U]) $\vee$ ($\exists$U,V,Z) (X1=U, X2=[V|Z] , on(U,Z))
person(X) $\leftrightarrow$ person(X),~male(X)
male(X) $\leftrightarrow$ X=tom $\vee$ X=bill

Program completion
Let P be a set of SLDNF clauses.
completed(P) comprises:
(i) the completed definition for every predicate that appears in the head of a clause in P,
(ii) the definition
Q(X1,..,Xk) $\leftrightarrow$ false k$\geq$o
for every k-adic predicate Q$\in$*P* for which there are no program clauses in P.

comp(P) = completed(P)$\cup$HET

Soundness results for SLNDF (Clark 1978)

Let T be the tuple of variables of query G and let SR be a safe computation rule.

R3.3.1 The SR computed answer S for goal G satisfies
comp(P) $\models$ $R^T{}_S \subseteq R^T{}_G$

R3.3.2 If using SR every branch of the computation tree routed at <G,{}> ends in failure then comp(P) $\models$ ~$\exists$G

R3.3.3 If the SR-computation tree routed at G is finite and S1,..,Sn, n$\geq$0, are all the answers computed by the success terminating branches, then
comp(P) $\models$ $R^T{}_G = R^T{}_{S1} \cup \ldots\ldots \cup R^T{}_{Sn}$,
or equivalently
comp(P) $\models$ ($\forall$X1,..,Xk) [G $\leftrightarrow$ $\exists$S1 $\vee$ ...$\vee$ $\exists$Sn]
where X1,..,Xk are the variables of T and $\exists$Si is Si existentially quantified with respect to every variable not in T.

Generalisations of SLDNF

Allowing explicitly quantified negated calls

R3.3.2 will allow us to generalize SLDNF programs to have negated existentially quantified conjunctions of literals of the form ~(∃Y1,..,Yk)C in place of negated atoms. As an example, we can allow a clause of the form:

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

More formally, a program clause can be an implication of the form

r(t1,..,tn) ← C1,..,Ck

where each Ci is a *body* call.

A *body* call is

(1) an atom

(2) a negated atom

(3) of the form ~(∃Y1,..,Yk)C where C is a conjunction of body calls.

The program clause is implicitly universally quantified with respect to all its free variables.

The state of the computation now contains a multiset of program calls. The safety condition on the computation rule is the condition that (1) a body call ~B can be selected only if B.S is ground and (2) a body call ~(∃Y1,..,Yk)C can be selected only if (∃Y1,..,Yk)C.S does not have any free variables.

Thus, ~(∃Y)(maths_course(Y),~takes(X,Y)) can be selected only when X is bound to a variable free term p. The evaluation of maths_course(Y),~takes(X,Y), with X=p, , will be a proof of

~(∃Y)(maths_course(Y),~takes(p,Y))

if it fails.

Implementations

NU-Prolog(Naish 1986) allows such negated existentially quantified conjunctions and enforces the generalized safety check that all free variables of such a negation are bound to ground terms in S before it can be selected.

Prolog allows negated conjunctions but these are not explicitly existentially quantified. It does not impose any form of safety check, using always the leftmost call rule. In consequence, a failed proof of any negated body call ~C is a proof of ~(∃Y1,..,Yk)C where Y1,..,Yk are all the variables of C.S at the time that the call is selected.

Allowing arbitrary conditions in queries and clauses

In (Clark 1978) an example was given of how SLDNF coupled with simple program transformation could be used as a scheme for logic programs which have much more complex conditions, such as universally quantified implications, in queries and clause bodies. The example given was the transformation of the goal student(X),($\forall$Y)[maths_course(Y) $\rightarrow$ takes(X,Y)]

into the query

student(X),~non_maths_major(X)

and additional program clause

non_maths_major(X) $\leftarrow$ maths_course(Y),~takes(X,Y)

for the introduced predicate non_maths_major. Alternatively, if we allow negated existentially quantified conjunctions as conditions, as suggested above, we can re-express the query as

student(X),~($\exists$Y)[maths_course(Y),~ takes(X,Y)]

Lloyd and Topor(1984) have taken up this idea and generalised it to allow program clauses of the form

A $\leftarrow$ W

where W is an arbitrary first order formula. By introducing extra predicates, they show how such a clause can be transformed into a set of equivalent SLDNF clauses.

Suspending the evaluation of a negated call with free variables

Another extension of the negation as failure rule is to allow selection of a negated call ~($\exists$Y1,..,Yk)C from a state <G,H> where ~($\exists$Y1,..,Yk)C.S has unbound free variables X1,..,Xn. The negation as failure rule is then modified so that its search strategy suspends any branch of the computation of <C,H> that tries to generate a non-variable binding for one of the free variables Xi. Xi becomes a *suspension* variable for <C,H>

If every branch of the evaluation of <C,H> terminates in failure without suspending, i.e. without trying to generate a binding for one of the free variables, we have proved ($\forall$X1,..,Xn)~($\exists$Y1,..,Yk)C.S. Hence we can simply delete the negated call from G as for negation as failure with a safe computation rule.

If there is a success terminating path for <C,H>, which also does not bind any free variable X1,..,Xn, we can add *false* to the H component of <G,H>. For we have proved ($\forall$X1,..,Xn)($\exists$Y1,..Yk)C.S. No matter what ground bindings S' are given to the variables X1,..,Xn, ($\exists$Y1,..Yk)C.(S$\cup$S') is **true**, hence ~($\exists$Y1,..Yk)C.(S$\cup$S') is **false**.

If every other branch suspends or ends in failure, we suspend the construction of the search tree for <C,H> and continue with the selection of some other call from state <G,H>. As soon as a descendant <G',H'> is generated in which H' has a non-variable binding Xi=t for some suspension variable Xi of <C,H>, the construction of the negation as failure search tree is resumed with the binding Xi=t added to the H component of every state of the partially constructed search tree for <C,H>. We now treat the negated call ~(∃Y1,..,Yk)C has though it had been selected in state <G',H'> and not the state <G,H>. The resumed evaluation may of course resuspend, but it might also terminate with some branch ending in success or with every branch ending in failure. In the former case, *false* is added to H', in the latter case, we delete ~(∃Y1,..,Yk)C from G' and continue with the evaluation from state <G'-{~(∃Y1,..,Yk)C},H'>.

Implementation

IC-Prolog allows negated calls to be selected which contain free variables and it interprets proof of the unnegated call which does not bind any of the free variables as a proof of falsity. However, it raises an error instead of suspending if there is an attempt to bind one of the free variables X1,..,Xn.

Allowing generation of values for the free variables

R3.3.3 will also justify a negation as failure rule which generates answers. Suppose we again drop the safety condition and allow any negated condition ~(∃Y1,..,Yk)C to be selected in a state <G,H> even if ~(∃Y1,..,Yk)C.S is not ground Let us now always construct the complete search tree for <C,H>, even if some of the paths generate bindings for the free variables of ~(∃Y1,..,Yk)C.S. As before, if there is a success branch that does not generate any bindings for these free variables, the call ~(∃Y1,..,Yk)C has failed. If all branches fail, ~(∃Y1,..,Yk)C has been proved true and we can delete it from <G,H>. But suppose there are exactly n success terminating branches ending in states <{},H1>, ...,<{},Hn> and S1,..Sn are the substitution equivalents of H1,..,Hn restricted to the tuple of bindings for the free variables of ~(∃Y1,..,Yk)C.S. A simple corollary of result R3.3.3 tells us that (∃Y1,..,Yk)C.S ↔ ∃S1v..v∃Sn. So, ~(∃Y1,..,Yk)C in state <G,H> can be replaced by the conjunction ~∃S1,...,~∃Sn. Distributing the negation through each ~∃Si will produce a disjunction of universally quantified inequations. As an example, suppose the program clauses for parent are:

parent(bill,john)
parent(joan,john)

A complete evaluation of parent(X,Y) will return the two sets of answers {X=bill,Y=john}, {X=joan,Y=john} allowing us to replace ~parent(X,Y) by

~(X=bill,Y=john),~(X=joan,Y=john)

or equivalently by

(X≠bill v Y≠john), (X≠joan v Y≠john)

These are inequation *constraints* on the bindings that can be given to X by the evaluation of any other call in which X appears.

To handle these inequation constraints, we need to extend the SLDNF scheme to a scheme for which the reserved set of predicates includes ≠, and for which unification is replaced by some process that can incrementally check solvability of sets of (possibly quantified) inequations and equations for the Herbrand interpretation. This is a non-unification based scheme which we shall consider 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

$p \leftrightarrow p, q \leftrightarrow p \vee \sim p$

of the program

$p \leftarrow p \quad q \leftarrow p \quad q \leftarrow \sim p$

but the SLDNF evaluation of query q will not terminate under any computation rule. This is because the evaluation cannot make use of the law of the excluded middle,

$p \vee \sim p$,

hence it cannot show that q is true no matter what the the truth of p is.

Intuitionistic logic similarly does not allow the use of the excluded middle law. In (Shepherdson 1985) the connection with intuitionistic logic was formalised by proving that the soundness result R3.1.1 holds if the logical implication $\models$ is replaced by intuitionistic provability.

This brings us to the issue of completeness of SLDNF. Clearly, SLDNF is not complete, in the sense that it can find the all the answers that are derivable from comp(P) for arbitrary programs P. The above is a counter example.

Completeness results for SLDNF

There is no simple completeness result, even for the simple version in which only negated atoms are allowed and the computation rule must be safe. The problems are threefold.

Firstly, there is no guarantee that for an arbitrary program that the computation will not flounder.

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 from <B,H> using the chosen computation rule.

The first two conditions force us to put extra constraints on the syntactic form of SLDNF programs, which we consider below. The last one requires that the computation rule used for each negation as failure proof be *fair* as well as safe.

A *fair* computation rule is a concept introduced into logic programming by Lassez and Maher (1984). 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 search tree is generated for call p(a). With any fair rule a finite search tree is constructed.

Hierarchical programs

Consider the directed graph representing the relation *refers to* for the predicates of program P. There is a +(-) labelled edge in the graph between predicates p and q if q appears in a positive(negative) call in the body of some clause about p in the program. An 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 Shepherdson (1985):

R3.3.4 *Completeness for hierarchical programs*
If the program is hierarchical, every variable in a clause occurs in a positive literal in the body, and every variable in a goal G also occurs in a positive literal of G, then any grounding substitution S for all the variables T of G satisfying comp(P) ⊨ G.S is SR-computable for any safe computation rule SR.

The conditions concerning variables ensure that the evaluation will not flounder and that every computed answer is a set of ground bindings for all the variables of G. (Clauses with no body conditions cannot contain any variables.) This is why we can require S to be a grounding substitution. The hierarchical condition ensures that every call generates a finite search tree, hence that everything can be inferred from comp(P) constructively.

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

A completeness result that is of considerable importance, even though it is only the base case of the general result that we would like, was given by Jaffar et al (1983):

R3.3.5 *Completeness for negation free programs*
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 the program we have already considered as a counter example of completeness

$p \leftarrow p \quad q \leftarrow p \quad q \leftarrow \sim p$

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

Completeness for structured programs

A completeness result that allows recursive definitions and negated calls is given by Barbuti and Martelli (1986) for *structured* programs.

A *stratified* program(Apt et al. 1988) is a program in which there are no cycles containing a - labelled edge in the *refers to* graph of the program. This allows nested negations and mutual recursion, but it does not allow recursion or mutual recursion through a negation.

A *structured* program is a stratified program P such that: if P' is the definite clause program that results from deleting all negated atoms in P, then for each ground atom A either comp(P') ⊨ A or comp(P') ⊨ ~A.

The stratification condition allows an inductive proof of the completeness result. The extra condition required for a structured program ensures that proofs from comp(P) are constructive. The problem is that it is far from being a simple syntactic condition on the program. The Barbuti and Martelli result is:

R3.3.6 *Completeness for structured programs*

Let P be a structured allowed program, SR a fair, safe computation rule and G an allowed goal and A a ground atom.

(1) Every grounding substitution S for the variables of G such that comp(P) |= G.S is an SR-computable answer.

(2) If comp(P) |= ~A, then every branch of the SR computation tree for A ends in failure.

Completeness for strict programs

The strongest completeness result, which like the Shepherdson result has the merit of having easy to check syntactic conditions on the program and query, is given by Kunen(1989) for *semi-strict* allowed programs.

A program P is *strict* with respect to some set of predicates Q iff in its *refers to* graph there is no pair of distinct predicates p,q ∈ Q such that there is one path from p to q which contains an even (possibly 0) number of - labelled edges and another path from p to q that contains an odd number of - labelled edges. In addition, for each predicate p∈ Q there is no path from p back to p with an odd number of - labelled edges.

A program is *strict* (Apt et al. 1988) iff it is strict with respect to all its predicates.

In a strict program a predicate p cannot be defined directly or indirectly in terms of positive and negative atoms for some other predicate q, or recursively through an odd number of negations.

A program is *semi-strict* (or *call consistent*) (Sato 1987) iff there is no predicate p in its refers graph for which there is a loop from p back to p with an odd number of - edges.

A program P is *strict with respect to goal* G iff P ∪ {q(X1,..,Xk) ← G}, q a predicate not in P, X1,.,Xk all the variables of G, is strict with respect to all the predicates of G. The Kunen completeness result is:

R3.3.7 *Completeness for semi-strict programs*
Suppose P is semi-strict and strict with respect to G and both are allowed.
(1) If comp(P) ⊨ G.S for some grounding substitution S, then S is an SLDNF computable answer.
(2) If comp(P) ⊨ ~∃G, then there is a finitely failed SLDNF tree for G.

The strictness conditions ensures that there *cannot* be a non-constructive proof from comp(P). In fact, it ensures that for every classical derivation of G.S from comp(P) there is derivation in 3-valued logic (in which the law of excluded middle is not valid). This is used to prove the Kunen result.
The syntactic constraint of strictness also guarantees consistency of completed(P) (Kunen 1989).

Strict completion
An interesting alternative approach to proving completeness, presented in (Drabent & Martelli 1989), is to change the reference theory which defines the answers that must be computed in order for SLDNF to be complete. They argue that a more natural reference theory is not comp(P) but comp*(P)=comp(split(P)) where split P is an SLDNF program that is derived from P by a program transformation. Split and comp* have the following properties:

R3.3.8 *Equivalence of P and split P*
split(G) evaluated with respect to split(P) has exactly the same set of computed answers as G evaluated with respect to P. split(G) has a completely failed search tree for rule SR iff G has a completely failed search tree for rule SR.

R3.3.9 *Properties of split(P)*
split(P) is strict (hence semi-strict) and split(P) is strict with respect to split(G).

These two results, together with R3.3.7 and R3.3.1 give us:

R3.3.10 *Completeness relative to comp*(P)*
For an allowed program P and query G and grounding substitution S, comp*(P) ⊨ split(G).S iff S is an SLDNF computable answer substitution for goal P and program P.

It remains to define the transformation *split*. For each predicate r in P, a new complement predicate r' not in P is invented. Then, for each clause
r(t1,..,tk) ← L1,..,Ln
in P, split(P) contains two clauses
r(t1,..,tk) ← pos(L1),..,pos(Lk)

r'(t1,..,tk) $\leftarrow$ neg(L1),..,neg(Lk)
where
pos(Li) = Li if Li is an unnegated atom
Li with its predicate p replaced by complement p' if Li is negated
neg(Li) = Li if Li is a negated atom
Li with its predicate p replaced by complement p' if Li is unnegated
For a goal G=L1,..Lm, split(G)=pos(L1),..,pos(Lm).

Example: For program P comprising the clauses
p $\leftarrow$ p
q $\leftarrow$ p
q $\leftarrow$ ~ p
split(P) is
p $\leftarrow$ p
p' $\leftarrow$ p'
q $\leftarrow$ p
q' $\leftarrow$ p'
q $\leftarrow$ ~ p'
q' $\leftarrow$ ~ p
comp(P) is p $\leftrightarrow$ p, q $\leftrightarrow$ p v ~ p
which is equivalent to p
comp*(P) is p $\leftrightarrow$ p, p' $\leftrightarrow$ p', q $\leftrightarrow$ p v ~ p', q' $\leftrightarrow$ p' v ~ p
which is equivalent to q $\leftrightarrow$ p v ~ p', q' $\leftrightarrow$ p' v ~ p

So the problem we had before, that p was derivable from comp(P) but not provable by any SLDNF computation does not arise with comp*(P).

The split transformation produces a program such that no proofs from comp(split(P)) can use the law of the excluded middle, hence the claim that this is a better reference theory for SLDNF computations from P which also cannot use this law.

Other work

Other completeness results are given in (Cavedon & Lloyd) and (Apt 1990). Shepherdson (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 - was the (Wolfram et al. 1984) GLD scheme.

Syntax

Programs, and goals are as in SLD.

Operational semantics

States of the computation are as in SLD. The difference is the computation rule. The GLD computation rule selects $n \geq 1$ calls G'={B1,..,Bn} from the G component of state <G,H>.

Let ri be the predicate of selected atom Bi and let n_{ri} be the number of program clauses about ri.

Let {A1←G1,..,An←Gn} be one of the $n_{r1}*n_{r2}*..*n_{rn}$ sets of program clauses (with variables renamed so each clause has a distinct set of variables and no clause has any variable in common with <G,H>) such that for $1 \leq i \leq n$ Ai has predicate ri. For each such set of clauses there is a next state which is

$<(G-G') \cup G1 \cup ... \cup Gn, H'>$

where H' is result of applying one of the unification algorithms to

H,E where E={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 unification of all the selected calls with the corresponding clause heads must terminate before goals can be selected for the next step. The reduction of each selected goal to the body of one of the clauses for its predicate is thus a synchronized step. The following generalization of GLD, implicitly described by Wolfram et al but not named, allows for asynchronous reduction of calls on different processors. It allows selection if one of the body calls from an introduced Gi before the unification of all the selected calls with the corresponding clause heads has terminated. It even allows selection of one of the introduced calls in the body Gi of clause Ai ← Gi before the unification of Bi and Ai has terminated. We shall call the scheme AGLD (for Asynchronous GLD).

To allow selection of a new call before all the unifications have terminated we must allow an AGLD state to contain equations of the form t=t' that have not yet been reduced to a set of bindings by applying the rules of some unification algorithm.

Operational semantics

An AGLD state is a triple of the form <G,H,E> where G is the multiset of calls, H is a Herbrand assignment which might also include *false* and E is a set of general term equations. The initial state is of the form <G,{},{}>. A success termination state is of the form <{},H,{}>, and a fail termination state is of the form <G,H∪{*false*},E>.

The AGLD computation rule selects $k \geq 0$ calls G'={B1,..,Bk} from G, as in GLD, but it also selects $n \geq 0$ equations E'={e1,..,en} from E, $k+n \neq 0$. Then, *one* step of one of the parallel unification algorithms of section 2 is applied to H,E using E' as the selected set of equations from E. Thus, if we are using algorithm 2, the unification rewrite rules are simultaneously applied to each ei ∈ E' subject to the restriction that we apply rule (c)(ii) to at most one equation X=t for each variable X. Any other equation X=t' in E' is a simply added back to E.

Let H',E" be the new H and E that result from this one step application of the unification rewrite rules to each of the selected {e1,..,en} from state <G,H,E>.

Let {A1←G1,..,An←Gn} be one of the $n_{r1}*n_{r2}*..*n_{rn}$ sets of program clauses (with variables renamed so each clause has a distinct set of variables and no clause has any variable in common with <G,H,E>) such that for 1≤i≤n Ai has predicate ri. For each such set of clauses

$$<(G-G')\cup G1\cup ...\cup Gk, H', E"\cup\{B1=A1,..,Bk=Ak\}>$$

is a next state of the computation.

The definition of computed answer is as for SLD.

Logical semantics

The results of the Wolfram et al. paper show that the set of computed answers is exactly the same as those computed by an SLD search tree for any AGLD computation rule.

Types of computation rule

GLD is AGLD with a computation rule that selects only equations until the E component of the state has been reduced to {} or *false* has been added to H.

At the other extreme, we can have a computation rule that only selects equations when the goal component of the state is {}, delaying all unification until the end of the computation. Wolfram et al. use this rule to prove the independence of the set of computed answers from the computation rule, which at this extreme is just a way of collecting together the final set of equations E to be reduced to solution form by application of an equation rewrite unification algorithm.

Consider now an intermediary computation rule which selects atoms along with equations but which delays the selection of any of the introduced atoms Gi from the body of the clause Ai ← Gi used for the i'th call Bi until some future state in which the equation Bi=Ai has been reduced to a Herbrand assignment Hi. A variant of algorithm 3 is used with each added equation Bi=Ai treated as an independent subset Ei={Bi=Ai} with its own local set of bindings Hi intialised to {}. When {Bi=Ei} has been reduced to {} the computation rule can select one or more of the calls in Gi. However, the bindings Hi are not at this stage added to E0 to be compared with the bindings Hj generated from some other call/head equation Bj=Aj. Instead, selected calls in Gi are unified with corresponding clause heads relative to H∪Hi with generated bindings added to Hi, and calls in Gj are unified with corresponding clause heads relative H∪Hj with generated bindings added to Hj. Only when there are no calls in the state which descend from Bi is Hi added to E0 to be compared with the bindings generated by the complete solution of some other call Bj. This corresponds to an implementation in which each of a set of multiple selected calls is initially evaluated independently as in SLD with the computed bindings for shared variables of the calls being compared as and when each call is *completely solved* (has been reduced to the empty set of calls and a Herbrand assignment Hi). The Epilog system of (Wise 1986) and the Prism system of (Kasif at al 1983) are or-parallel implementations of AGLD (strictly a slight variant of AGLD) with this computation rule.

A sequential AGLD computation rule is one which at each step selects a single equation or a single atom to replace. If the rule always selects equations until E is {}, 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=Ai pursued until a binding X=t, t a non-variable is generated. This binding is retained in E, and not moved to H. Some other call, perhaps the designated producer of X is then selected in order to generate a binding X=t', t a non-variable, which is added to H. The computation rule can now switch back to the suspended unification rewrite of Bi=Ai using rule (c)(i) to replace X=t by t'=t.

Absys (Foster & Elcock 1969), which can with some justification be considered the first logic programming language (see Elcock 1990), 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 section 3.3. The computation rule never selects an equation of the form X=Y, X and Y distinct variables. Such variable/variable equations remain in E. A success state of the computation is a state <{},H,E> in which H is a Herbrand assignment and E={} or is a set of variable/variable binding equations of the form X=Y. Each X=Y left in E is thus a qualification to the computed answer.

Data flow computation rules

Let us assume that steps of unification algorithm 2 are being applied to the selected set of equations E'. For SLD, we discussed the idea of a data flow computation rule which selected another call, the designated producer of X, if the unification of a call Bi with the head of one of its clauses Ai would generate a binding equation X=t. X is treated as an *input* variable of Bi and its selection must be delayed until a binding X=t' has been genereted by the producer.

In AGLD, the analogue of this sequential data flow rule, is a parallel computation rule which will leave a binding equation X=t generated by rewriting Bi=Ai in E if X is an input variable of call Bi. That is, the computation rule cannot select this equation for transfer to H. Transfer to H would amount to communication of the binding to all other calls. Instead, the equation will remain in E until another binding equation X=t' is added to H by the unification rewrite of some other call/head equation for which X is not a designated input variable of the call. At that point, X=t can be selected for application of rule (c)(i). Bindings generated by the rewrite of Bi=Ai for variables not designated as input variables of Bi can be transferred to H.

If need be, we can distinguish between occurrences of variables, preventing transfer of binding equation X=t from Ei to Hi only if the binding 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 by the rewrite of Bi=Ai for non-input variable occurencies of Bi can be globally broadcast by being transferred from E to H. We shall call these the *allowed* bindings for the call.

Atomic unification

Suppose that steps of algorithm 4 are used instead of algorithm 2. That is, E is divided into subsets E1,..,Ek where each Ei is a set of equations resulting from the rewrite of some call/head equation Bi=Ai that was added to E. In addition, each Ei has an associated Hi of local bindings. This algorithm does not transfer binding equations directly from E to H, instead all bindings go first to Hi. The same constraint on bindings for input variables of the call Bi can apply. No equation X=t generated and added to Ei for an input variable occurence of X in Bi will be transfered to Hi. Instead this will be left in Ei until some binding X=t' is added to the global set of bindings H. For algorithm 4, bindings can transferred from Hi to H when E is empty. Then, providing H does not by now contain bindings for variables bound in Hi, all the bindings in Hi must be selected and transferred atomically to H. This is *atomic unification* of Bi and Ai.

If atomic unification is coupled the extra constraint that no call in Gi is selected until Ei and Hi are both {}, this is *atomic test unification.*

In a multiprocessor implementation, atomic unification requires synchronization of the global broadcasting of variable bindings. A given processor must get binding permission for each of the variables bound in Hi before broadcasting the bindings. It must be prepared to relinquish the given binding permissions if it cannot get binding permission on them all. See (Taylor 1989) for a discussion of atomic unification on a multiprocessor.

Programming language features for specifying the input variable occurrences of 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. That is, suppose variable V occurs in the binding term t' of a binding X=t' added to H (by the evaluation of some other call), where X is an ? annotated variable of the call Bi. V is not also an input variable of Bi, unless it is also annotated with a ?. If the input variable property was inherited, making V an input variable of Bi whether or not it was annotated with ?, and in addition V became an input variable to all calls that descended from Bi, we would have the data flow analogue of the IC-Prolog eager consumer.

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

In the Relational Language (RL) (Clark & Gregory 1981) and Parlog (Clark & Gregory 1986, Gregory 1987), this is done by specifying an input ? or output ^ mode for each argument for each k-adic program defined predicate r . These languages are both instances of AGLD using unification algorithm 2.

Let r(t1,..,tk) be the head of the clause Ai $\leftarrow$ Gi and let r(t'1,..,t'k) be the call Bi. If m is in an input argument position, then only bindings generated for variables in the clause are allowed bindings for the unification rewrite of tm=t'm. If the rewrite of tm=t'm generates an equation V=t for any other variable, V is treated as an input variable, and the binding cannot be transferred from E to H. Thus, the unification for input argument terms is just input matching; for it is not allowed to generate any binding for a variable not in the clause. We can actually compile the input argument terms of the head of a clause into code that will do the input matching and automatically suspend when it wants to match a variable not in the clause to a non-variable term (see Gregory 1987)

If n is an output argument position, all bindings generated by the rewrite of tn=t'n are allowed and can be transferred to H as soon as they are generated. Bindings transferred to H for variables not in the clause are the*output* bindings for the unification. In RL and Parlog, the rewrite of equation tn=t'n is not started, that is the computation rule cannot select the equation, until all the equations tm=t'm for the input argument positions have been reduced to a set of allowed bindings that have all been transferred to H. At this point, calls in Gi can also be selected. That is, the rewriting of the equations for the output argument terms can proceed in parallel with the selection of the body calls of the clause.

In GHC (Ueda 1985), which is very similar to Parlog, for the whole unification rewrite of Bi=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 = calls in the body of the clause.

Guard calls

Parlog and GHC both allow the programmer to specify some subset T'i of the body calls Gi as *guard calls*. Guard calls can be selected immediately, i.e. they can be evaluated concurrently with the rewrite of the input argument equations. Restrictions with respect to the transfer of bindings from E to H also apply with respect to these guard calls. Only variables of the clause Ai $\leftarrow$ T'i,G'i and variables of clauses used to evaluate calls in T'i can be transferred form E to H. This is the *safety* condition for the guard calls. In Parlog, the program can be checked at compile time to ensure that guard calls are safe (see Gregory 1987). This compile time check guarantees that any binding equation they add to E for a variable that cannot be transfered to H will have been added to E by an input match, hence will automatically be left in E. In GHC, guard calls can be defined by programs that may add non-transferable bindings to E by output unification, so GHC must have a runtime test to prohibit such bindings being transferred to H, as they normally would be. This is the main difference between the languages. In Flat Parlog and Flat GHC only calls to primitives, i.e. non-program defined relations can appear in the guard set of a clause. Flat Parlog and Flat GHC evaluate programs in exactly the same way.

In both languages, the rewrite of the equations for the output argument positions of the call Bi and the selection of calls in G'i is delayed until (1) no call in T'i, and no descendant of such a call, appears in the goal component G of the state of the computation *and* (2) no input argument equation tm=t'm of call Bi or a descendant, and no equation added by the evaluation of a guard call, or a descendant, appears in the E component of the state.

Note that this restriction on what bindings can be transferred to H during the input matching and guard call evaluation means that we can in parallel unify with the clause heads and evaluate the guard calls of all the clauses for Bi *without* trying to add generating competing bindings for the same variable to H. All the bindings that each of these alternative evaluations will add to H are for local variables of the clauses that they use. This is important. It allows us to keep a single location for each broadcasted binding even though there is or-parallel evalaluation of the input matching and guard calls of the alternative clauses. Usually an or-parallel evaluation needs to keep multiple alternative bindings for variables, which either delays access or requires copying of the goal components of the state.

Committed choice

Suppose that there is a state of the computation in which the unification rewrite of all the input argument equations for call Bi has succeeded and the evaluation of all the guard calls T'i have successfully terminated producing a set of allowed bindings all of which have been transferred to H. In the Relational language (the first committed choice language), and in Parlog and GHC, the evaluation *commits* to the exclusive use the clause Ai ← T'i,G'i for call Bi. All competing parallel unifications and guard evaluations using other clauses for the call can be aborted.

In Concurrent Prolog, which also has guard calls but which uses unification algorithm 4, there is no commitment to the clause Ai ← T'i,G'i and no call in G'i is selected, until Bi=Ai has been completely reduced to a set of allowed bindings that have been transferred to the local Hi *and* all of these bindings have been successfully atomically transferred to H. Note that in Concurrent Prolog, competing bindings for the same output variable of the call must be kept in the different local Hi. In Parlog and GHC competing output bindings for a call are never generated because the output argument unifications that generate these bindings are only started after the commitment to use just one clause for the call.

In programming terms there is a disadvantage to this delay in generating the output bindings until after the commitment to use a clause. In Parlog and GHC one must program in such a way that only one call will generate a binding for each variable, with all other calls suspending until that binding is broadcast, or all the competing calls must generate compatible (unifiable) bindings. In Concurrent Prolog, one can allow calls to compete, with the atomic test unification making sure that *only one* binding is globally broadcast (added to H) *and* that all other calls test the broadcasted value before committing to a clause. The disadvantage is the complexity of the implementation of atomic unification and the need to record multiple bindings.

(Burt & Ringwood 1988) have recently proposed a simpler notion of atomic test broadcasting of a single ouput binding. A single output binding is designated as the test binding. The computation rule must select this designated output binding in E, test that there is no binding for the variable already in H, and transfer the binding to H at the point of commitment to use the clause.

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 commitment 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 1989) 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, selects in each state all calls Bi for which there is only one clause with a head Ai which will unify with Bi in the context of the global bindings H. To implement this, heads of the different clauses must be unified with Bi in parallel with bindings for call variables generated and added to an Hi which is specific to the clause. If more than one Bi/head unification succeeds, the call Bi is suspended until bindings are added to H for one or more variables bound in some local Hi. The parallel unifications of Bi are resumed to take into account these new bindings. If they cause all but one of the unifications to fail, the call is reduced to the body of the only unifying clause and all the equations of its Hi can be globally broadcast by being addded to the H component of the state. P-Prolog (Yang & Aiso 1986) has such a computation 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 (Haridi & Brand 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.

The CP language proposal 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.

5 Schemes based on general equation solving

5.1 Equation solving over rational trees - Colmerauer's RTS scheme

Nearly all the implementations of the SLD or SLDNF schemes do not implement the occur check of the unification algorithm. This means that assignments are generated which are not Herbrand assignments. We can, for example, have an assignment X=f(a,X) which denotes the infinite tree f(a,f(a,f(a,.....))).

If we allow general assignments as answers, we are not solving equations for the Herbrand interpretation. We are solving them for the domain of infinite rational trees, i.e. the domain of infinite trees each of which can be finitely represented by an assignment.

Unification, or equation solving, for infinite trees has been independently studied by Huet (1976) and others but to my knowledge Colmerauer (1982) was the first to propose a logic programming scheme based on such equation solving. Colmerauer did not name his scheme. We shall call it RTS, for *rational tree* scheme.

Syntax

For RTS, the reserved predicates and the syntax of programs and queries is the same as SLD.

Operational Semantics

States of the computation are pairs <G,A> where G is a multiset of calls and A is an assignment. The initial state is of the form <G,{}>, where G is the multiset of calls from the query. A success termination state is of the form <{},A> with computed answer A restricted to equations which contain at least one variable Y such that some variable Xi of the given query depends on Y in A. See section 1.1.

. A failure termination state is of the form <G,A> where *false* $\in$ A.

For non-termination state <G,A> a next state of the computation is generated by selecting one of the calls r(t1,..,tk) in G. Let r(t'1,..t'k) $\leftarrow$ G' be some program clause for its predicate (with variables renamed as usual). A next state is <G-

{r(t1,..,tk)}∪G',A'>, where A' is the result of applying the following equation rewrite algorithm to the pair A, E={t1=t'1,..,tk=t'k}.

Colmerauer's equation rewrite algorithm

We give a modification of Colmerauer's algorithm that relates it more to our unification algorithm 1. |t| denotes the number of variables and functors in the term t. It is a measure of the size of t.

Repeat the following step until E={} or A contains *false*. On termination return A as the answer.

Select any equation e in E and *delete* it from E.
Cases:
(a) e of the form f(t1,..,tk)=f(t'1,..,t'k). Add the k equations t1=t'1,..,tk=t'k to E.
(b) e of the form X=X. Do nothing.
(b') e of the form X=Y, X and Y distinct variables.
  Subcase (i) X=t $\in$ A. Add t=Y to E.
  Subcase (ii) No X=t $\in$ A. Add X=Y to A.
(c) e of the form X=t, or of the form t=X, where t is not a variable.
  Subcase (i) X=t' $\in$ A and |t'|≤|t|. Add t'=t to E.
  Subcase (ii) X=t' $\in$ A and |t|<|t'|. Add t=t' to E and replace X=t' in A by X=t.
  Subcase (iii) No X=t' $\in$ A. Add X=t to A.
(d) e of the form f(...)=g(...), where f and g are different. Add *false* to H.

Logical Semantics

What is the relationship of the assignment A' produced by this algorithm to the initial A∪E. For the unification algorithms we have the result R2.1. For Colmerauer's algorithm we have a similar result but for a different equality theory, which we shall name RTET (Rational Tree Equality Theory). RTET is the set of axioms of HET minus axiom schema (F3), the schema X≠t[X].

R5.1.1 RTET $\models R^T_{A\cup E}=R^T_{A'}$
Hence
R5.1.2 *false* $\in$ A' iff RTET $\models R^T_{A\cup E}$ = **false**

However, we cannot also conclude the analogue of R2.3
A' is a assignment iff RTET $\models R^T_{A\cup E}$≠**false**
because an assignment does not denote a non-empty relation for every interpretation. To get the analogous result, which confirms that A' is a solution for the set of equations A∪E for the domain of infinite rational trees, we must strengthen RTET with axioms that tell us that every assignment does denote a non-empty relation for this domain.

Following (van Emden and Lloyd 1984), the simple way to do this is to add the axiom scheme $\exists$A , A any assignment, to RTET to give the theory $\exists$RTET. $\exists$RTET is a first order theory of the infinite rational trees described in Colmerauer (1982). An infinite rational 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 the result:

R5.1.3 A is a assignment iff $\exists$RTET $\models$ $R^T{}_{A\cup E}$$\neq$**false**

Soundness of the Colmerauer scheme
R5.1.4 (van Emden and Lloyd 1984)
For every RTS computed assignment A for goal G and program P,
$\exists$RTET, P $\models$ $R^T{}_A \subseteq R^T{}_G$, $R^T{}_A \neq$ **false**

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

Extending RTS to RTSNF by including 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 for SLDNF we replace comp(P) by RTET$\cup$completed(P) to get the corresonding results for RTSNF. Appropriate versions of the completeness results of section 3.3 should also apply. In particular, since RTS is a special case of the scheme GLDE that we are about to describe. For GLDE the analogue of R3.3.5 holds, so this completeness result for negation free programs also holds for RTSNF.

5.2 Equation reduction using an general equality theory - Jaffar, Lassez, Maher GLDE scheme

Colmerauer's scheme is SLD with the Herbrand equality theory replaced by the theory RTET. In other words, the unification of SLD is replaced by equational solving (generalised unification) for theory RTET.

Jaffar et al (1986) present a generalization of SLD and RTS in which the particular theory, HET or RTET, can be any equality theory *E* satisfying a property they call *unification completeness*. In this scheme, the ordinary unification of SLD is replaced by *E*-unification, unification or equation solving relative to the theory *E*. Replacing unification by *E*-unification in a resolution system is an idea due to Plotkin(1972). The property of unification completeness required by Jaffar et al is the analogue of the key property R2.1 of unification. The scheme is actually a generalisation of GLD, hence we shall call it GLDE.

Properties of the theory E
Let E be a set of term equations containing variables T.
E is *E-unifiable* if there is an assignment A such that $E \models R^T{}_A \subseteq R^T{}_E$.

Generally, there will be many assignments A satisfying this condition, possibly an infinite number. There may or may not be maximally general assignments, the analogue of the mgu.

The equivalent of property R2.1 for theory *E* can be expressed as:

$E \models R^T{}_E = R^T{}_{(A1 \cup \ldots \cup Ai \cup \ldots)}$

where A1....Ai...... are all the *E*-unifiers of E. The $\supseteq$ inclusion follows from the definition of an *E*-unifier. The $\subseteq$ inclusion is the condition of interest.

E is *unification complete,*

if $E \models R^T{}_E \subseteq R^T{}_{(A1 \cup \ldots \cup Ai \cup \ldots)}$, where A1,..,Ai,..are all the *E*-unifiers of E.

When there are no *E*-unifiers, unification completeness enables us to conclude that

$E \models R^T{}_E = \textbf{false}$

This is the property needed to justify negation as failure and to prove R5.2.2 stated below.

Syntax

As for SLD, the set of reserved predicates *R={false, true, =}*

A program clause is an implication of the form

H $\leftarrow$ E,G

where E is a conjunction of equality atoms (term equations) and G is a conjunction of atoms with predicates from *P*. H is an atom with predicate from *P*. A query is the same form E,G as a clause body.

Operational Semantics

A state of the computation is a three-tuple of multisets <E,G,A> where E is a multiset of equality atoms, G is a multiset of a non-equality atoms and A is an assignment which may contain *false.*

The initial state for goal E,G is the state <E,G,{}>. A success termination state is of the form <{},{},A>. The computed answer is the subset of A *related* to the query as in RTS. A failure termination state is any state with *false* in A.

For a non-termination state <E,G,A> the computation rule selects a subset E'={e1,..,em} of E, a subset G'={B1,..,Bn} of G, $m+n \neq 0$. Let

H1 $\leftarrow$ E1, G1 .... Hn $\leftarrow$ En,Gn

be n variants of program clauses with no variables in common with each other or with the state <E,G,A> where the i'th clause is about the predicate of the i'th selected atom Bi. Let A' be an *E*-unifier of E'$\cup${B1=H1,..,Bn=Hn} in the context of assignment A. That is, A' is such that $E \models R^T{}_{A \cup A'} \subseteq R^T{}_{A \cup E' \cup \{B1=H1,..,Bn=Hn\}}$.

For each different set of n program clauses and for each *E*-unifier A', there is a next state of the computation of the form:

<(E-E')$\cup$E1$\cup$..$\cup$En, (G-G')$\cup$G1$\cup$...$\cup$Gn, A$\cup$A'>

If there is no such A', the next state has *false* added to the assignment component A.

Logical Semantics

Jaffar et al. (1986) prove the following soundness and completeness result:

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

R5.2.2 If B is a ground atom, completed(P),$E \models$ ~ B iff for a fair computation rule every branch of the search tree 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 of 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 *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. Jaffar et al (1987) show that Colmerauer's RTS is an instance of the scheme, hence the negation as failure completeness result R5.2.2 applies to RTS.

Implementable instances of the GLDE must of course have a theory *E* such that there are a finite number of *E*-unifiers and an algorithm computing all the *E*-unifier solutions of the set A∪E.

5.3 Equational Logic Programming

For an in depth treatment of equational logic programming see (Holldobler 1990). Here we shall briefly survey some of the results and proposed language schemes that are instances or variants of GLDE.

E-unification for equational theories

A *equational theory* is a theory *E* comprising sets of equations of the form t=t'. For certain equational theories algorithms can be given for generating all the *E*-unifiers of a pair of terms. Such an algorithm can then be used to generated *E*-unifiers of any set of equations {t1=t'1,..,tk=t'k}. We simply apply the algorithm to the pair tuple(t1,..,tk), tuple(t'1,..,t'k) where tuple is a functor not in any equation in *E*.

Gallier and Raatz(1989) present a scheme they call SLDE, which is a special case of GLDE. Only one call or equation is selected and *E*-unification is relative to a set of equations for which there is a complete procedure for generating all unifiers.

For a general survey of results on *E*-unification for equational theories see (Siekmann 1984). Below we shall consider only the case where the equations define a canonical term rewriting system. For such systems, unification can be fused with term rewriting to give an algorithm for generating all *E*-unifiers. There are also specialised equation rewrite algorithms, extensions of the Martelli and Montanari(1982) unification algorithm, where *E* is a canonical term rewriting system.

Rewrite systems

A set of equations *E* defines a *term rewrite system* if for each equation t=t' in *E*, t is not a variable and t' contains no variable not in t.

A *reduction* relation -> associated with *E* is defined as follows.
Let T, T' be two terms. T -> T' if
(1) t=t' is an equation in *E*.
(2) s is a subterm of T.
(3) A is an assignment
(4) s is identical to t.A and
(5) T' is T with s replaced by t'.A.

Example: T=g(h(X)), T'=g(j(X)), equation is h(Z)=j(Z) and A={Z=X}.

A *narrowing* relation => associated with *E* is defined as follows.
Let T, T' be two terms. T => T' if
(1) t=t' is an equation in *E* with variables renamed so that there is no clash with a variable in T.
(2) s is a subterm of T.
(3) A is an assignment
(4) s.A is identical to t.A and
(5) T' is T.A with s.A replaced by t'.A.

Example: T=g(X), T'=g(j(Z)), equation is h(Z)=j(Z) and A={X=h(Z)}.

Let ->* be the transitive reflexive closure of ->.
-> is *terminating* if there is no infinite sequence t1 -> t2 -> ..... -> ti -> .....
-> is *confluent* if whenever t ->* s, and t ->* s' there is a t' such that s->*t' and s'->*t'.
A term t is *canonical* for -> if there is no t' such that t->t'.
-> is *canonical* if it is terminating and confluent.

R5.3.1 If -> is canonical then for every term t, every reduction sequence starting a t leads to a unique canonical form for t.

Fay(1979) gives an algorithm, later improved by Hulot(1980) and (Jouannaud et al 1983) for generating a complete set of *E*-unifiers for a pair of terms t,t' when *E* is a set of equations that define a canonical rewrite system. The algorithm is essentially a systematic enumeration of the assignments A which are non-deterministic generated as follows. Let pair be a functor not in *E* or t,t'. Iteratively generate a narrowing chain pair(t1,t'1)=>......=>pair(tn,t'n) such that t1=t,t'1=t' and tn unifies with t'n with assignment An. If Ai, 1≤i≤n-1 is the assignmement used at the step pair(ti,t'i) => pair(t(i+1),t'(i+1)), then the composition A1.A2....An is an *E*-unifier of t,t'.
EQLOG (Goguen & Meseguer 1986) is a language/scheme in which this procedure is used to generate *E*-unifers during the computation of a program that comprises a set of

(many-sorted) definite clauses and a set of rewrite equations. EQLOG actually allows conditional rewrite equations. These are equations of the form e ← e1,..,ek where e and each ei are equations but variables of e1,..,ek are restricted to variables from e. Procedures for generating all the *E*-unifiers for canonical conditional rewrite equations have been given by (Kaplan 1986) and (Holldobler 1988).

Special equation rewrite algorithms

As an alternative to the above general procedure based on narrowing, Martelli at al (1986) proposed using a modification of the Martelli and Montanari unification algorithm for doing *E*-unification where *E* is a set of canonical rewrite equations. Another such algorithm is given in (Holldobler 1987).

Narrowing as an alternative step in the computation

Yamamoto(1987) proposed a scheme in which *E*-unifiers are not explicitly generated. Instead, narrowing, unification of = calls, and call reduction using a homogeneous form program clause are alternative ways of generating a next state of the computation.

Syntax

Programs for this scheme comprise homogeneous form clauses, a set of canonical rewrite equations and the single non-rewrite equation X=X.

Operational semantices

States are as in SLD.

The computation rule selects a call *or* a term within some call in the state <G,H>.

If it is a call r(t'1,...,t'k) to a program defined relation, let

r(X1,..,Xk) ← X1=t1,..,Xk=tk,B1,..,Bm

be one of the homogeneous form clauses for its predicate (variables renamed as usual). A next state is

<G-{r(t'1,..,t'k)}+{X1=t1,..,Xk=tk,B1,..,Bm},H∪{X1=t'1,..,Xk=t'k}>.

If the selected call is an equation t1=t2, unification algorithm 1 is applied to H,{t1=t2} to produce H'. The next state is <G-{t1=t2},H'>.

Finally, if a term s is selected within a call B, this is replaced by a new term t' by narrowing with one of the rewrite equations t=t' to give a new call B'. The term t is unified with s in context H to give H'. The next state is <G-{B}+{B'},H'>.

Logical semantics

Yamamoto(1987) gives soundness and completeness results.

Compiling equations into clauses

A final alternative is to do away with narrowing altogether and to emulate it using a normal SLD derivation for a 'compiled' form of the program in which equations are replaced by clauses. A result concerning such emulation of narrowing for ground terms is given in (van Emdem & Yukawa 1987). K-LEAF (Giovanetti et al 1989), is a equational logic programming language that implements narrowing by compiling equations to definite clauses.

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 equation solving algorithm for the RTS scheme was extended to apply to sets E,I of equations and inequations. Programs and goals were correspondingly extended to allow use of ≠. This scheme, which we shall call IRTS (for Inequation RTS) is the abstract model for Prolog II (Colmerauer 1986).

Syntax

For IRTS, the set of reserved predicates $R=\{true, false =, \neq\}$.

Programs are sets of definite clauses of the form

r(t1,..,tk) ← E,I,G

where G is a conjunction of atoms with predicates from *P*, E is conjunction of equations and I is a conjunction of inequations of the form t≠t'. The clause is about the predicate r and is implicitly universally quantified for every variable. A goal, has the form of a clause body.

A *reduced form* set of inequations only contains inequations of the form Y≠t or tuple(Y1,..,Ym)≠tuple(t1,..,tm) where Y1,..,Yk are distinct variables.

An *RT-consistent* set of equations and inequations is an assignment A and a reduced form set of inequations RI such that no variable appearing in the left hand side of some inequation i ∈ RI is bound in A.

Operational semantics

A state of the computation is a triple <G,A,RI>. G is a multiset of atoms with predicates from *P*. A is an assignment and RI is a reduced form set inequations where A∪RI is an RT-consistent set, or it contains *false.*

The initial state for a goal E,I,G is <G,A',RI'> where A',RI' is the RT-consistent set produced by applying Colmerauer's equation/inequation solving algorithm given below to E,I. The A,RI of the algorithm are both initialized to {}.

A success termination state is any state of the form <{},A,RI> where A∪RI is an *RT*-consistent set and a fail termination state is any state of the form <G,A,RI> where *false*$\in$ A$\cup$RI. The answer returned from success state <{},A,RI> is A,RI restricted to equations and inequations which contain at least one variable Y such that some variable Xi of the given query G,E,I depends on Y in assignment A.

For each non-termination state <G,A,RI>, some atom r(t'1,..,t'n) in G is selected using a computation rule. For each program clause r(t1,..,tn) $\leftarrow$ E,I,G' (with variables suitably renamed) there is a next state of the computation <G-{B}$\cup$G',A',RI'> where A',RI' is the result of applying the algorithm below to E$\cup${t1=t'1,..tn=t'n}, I in the context of the RT-consistent set A$\cup$RI.

Colmerauer's equation/inequation solving algorithm

The algorithm applies to a set E of equations and a set I of inequations in the context of an assignment A and a reduced set of inequations RI where A$\cup$RI is RT-consistent.

A,E is first reduced to an assignment A' or to a set of equations containing *false* using the algorithm of the RTS scheme.

If an assignment A'=A$\cup$A" is generated, let

T = I$\cup${i: i $\in$RI and some left hand side variable of i is bound in A"}.

For each inequation s$\neq$t $\in$ T, the RTS algorithm is reapplied to A'$\cup${t=s}.

If this produces *false*, the inequality t$\neq$s is discarded (because it is satisfied by assignment A'). If the algorithm successfully terminates without generating any extra bindings for variables, t$\neq$s is replaced by *false* in I (because the absence of bindings for the variables in t=s shows that t=s is satisfied for assignment A' for all rational tree values for these variables, hence there is no rational tree assignment for these variables that will satisfy t$\neq$s and equations A').

If t=s is reduced to a set of bindings Y1=t'1,..,Ym=t'm then t$\neq$s is replaced by the single inequation tuple(Y1,..,Ym) $\neq$ tuple(t'1,..,t'm) (by Y1$\neq$t1 if m=1) in I (because tuple(Y1,..,Ym) $\neq$ tuple(t'1,..,t'm) iff for some i Yi$\neq$t'i iff t$\neq$s). The result of the algorithm is therefore either a set of equations and inequations containing *false* or A',I' where A' is an assignment and 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', so A'$\cup$I' is an RT-consistent set.

Logical semantics

The theory RTET of the RTS scheme also validates this algorithm, we have:

R6.1.1 RTET $\models$ $R^T_{E,I} = R^T_{A',I'}$, where T is the tuple of variables of E,I.

Colmerauer shows that an RT-consistent set A'$\cup$I' always has a solution in the domain of infinite rational trees, hence the name. The theory $\exists\neq$RTET, which is RTET augmented with an axiom scheme $\exists$A,I , where A,I is any RT-consistent set of equations and inequations, gives us the correctness result:

R6.1.2 *Correctness of IRTS*
If $A \cup I$ is a computed answer for goal E,I,G containing variables T using program P then
$P, \exists{\neq}RTET \models R^T{}_{A,I} \subseteq R^T{}_{E,I,G}, R^T{}_{A,I} \neq \text{false}$

A strong completeness result for IRTS is given by R7.2 below, since IRTS is a special case of the CLP scheme of that result.

Implementation

In an implementation of the scheme, the inequations can actually be handled by a special negation as failure rule that returns bindings. For each inequation, $s \neq t$, an attempt is made to establish $s \neq t$ by trying to show that s=t fails in the environment A'. If s=t fails, $s \neq t$ is deleted. If s=t succeeds, without binding any variables in s or t, $s \neq t$ is replaced by *false*. If it succeeds generating bindings Y1=t'1,..,Ym=t'm for variables in the equation, we have proved that

$RTET \models R^T{}_{A \cup \{s=t\}} = R^T{}_{A \cup \{Y1=t'1,..,Ym=t'm\}}$

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

$tuple(Y1,...,Ym) \neq tuple(t'1,..,t'm)$

is returned as the 'answer' for the negated call $s \neq t$. 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 $s \neq t$ in the context of assignment 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 an answer returned by a negated call ~B with variables. The answer is the complement of the disjunction of all the answers returned by the unnegated call. In general, the complement contains inequations for the variables of B. These inequations are then checked for consistency relative to any subsequent bindings generated for the variables in a manner similar to that of the IRTS scheme. Thus SLDNF allows early selection of negated calls, and even returns final answers that contain inequation constraints as well as equational bindings. The answers are always consistent for the Herbrand interpretation, they denote non-empty relations for this interpretation, as we would expect.

Syntax

R={true, false, =, ≠}.

Program clauses and goals are as for IRTS except that the inequation calls in the bodies of clauses and goals can be universally quantified for some (or all) of their variables. Also, any call to a program defined predicate can be negated as in SLDNF. The clauses are implicitly universally quantified for their free variables.

Operational semantics

A state of the computation is of the form <G,H,E,QI> where G is a multiset of calls to program predicates, H is a Herbrand assignment, E is a multiset of equations and QI is a multiset of quantified inequations. H or QI can also contain *false*.

The initial state for a goal E,QI,G is <G,{},E,QI>.

A success termination state is of the form <{},H,{},PI> where PI is a set of primitive inequations for H. An inequation (∀V1,..,Vn)t≠s is primitive for H if it is HI consistent with all the bindings given in S, the substitution equivalent of H, for its free variables.

The computed answer is a *normalised* form of S'∪PI, where S' is S restricted to the free variables of the query. The normalisation process removes irrelevant inequalities (see Chan 1988 for details). In the normalised form of S'∪PI each variable in an equation, and each free variable in an inequation, is either free in the query, or it appears inside a non-variable binding term of an equation in S'. So, where there are no function symbols in the bindings of S', every variable in a normalised answer is a free variable from the query.

A fail termination state has *false* in H or QI.

The computation rule for SLDNF can select any call Bi in the goal G, any equation t=t' in E, or any inequation i in QI of state <G,H,E,QI>. The rule does not need to be safe.

A selected equation is handled by applying algorithm 1 to H,{e} to produce H'. The next state is <G,H',E-{e},QI>.

A selected inequation (∀V1,..,Vn)s≠t is tested to see if it is **true** or **false** in the context H by a special negation as failure rule. Unification algorithm 1 is applied to H∪{s=t} after variables V1,..,Vn have been renamed so that they do not clash with any free variable of the state.

The inequation is determined to be **true**, and deleted from the current set of inequations QI, if algorithm 1 generates *false* for H∪{s=t}. (We have established that HET ⊨ ~∃(s=t).S, S the substitution equivalent of H, hence that ∀(s≠t).S.)

It is replaced by **false**, if the algorithm produces an assignment H' that has additional bindings only for the renamed quantified variables V1,..,Vn. (This is one of the extensions to the negation as failure rule we discussed in 3.3.)

Unlike IRTS, the inequation is not reduced to another inequation if the algorithm succeeds but generates bindings X1=t1,..,Xk=tk for the free variables of (∀V1,..,Vn)(s≠t).S. In this case, (∀V1,..,Vn)s≠t is considered a *primitive* inequation for H and left unchanged in QI. We have shown that (∀V1,..,Vn)(s≠t).S is **false** for assignment X1=t1,..,Xk=tk to its free variables, hence it must remain as a constraint that to be rechecked if H is extended with any bindings for these variables. However, (∀V1,..,Vn)s≠t is consistent with H because it will be true for any extension of H which is HI inconsistent with the assignment X1=t1,..,Xk=tk.

If a positive call B is selected, let A ← E',QI',G' be a clause for its predicate (with variables suitably renamed). A next state of the computation is <G-{B}+G',H',E∪E',QI∪QI'> where H' is the result of applying algorithm 1 to H∪{B=A}.

If a negative call ~B is selected from G, a new auxiliary computation with initial state <{~B},H,{},{}> is commenced.

If every branch of the search tree ends in failure, the next state of the main computation is <G-{~B},H,E,QI> as with SLDNF.

If the search tree is finite, but contains success branches, the set of normalised answers {N1..,Nk} given by the finite number of success terminating branches is returned. This is negated and converted into an HET equivalent set {(E1,QI1),...,(En,QIn)} representing the set of different answers to ~B consistent with the current assignment H.

Each (Ei,QIi) is a set of equations Ei and quantified inequations QIi. Chan gives a procedure for generating this form as a representation of the complement of { N 1 , . . . , N k } .

For each (Ei,QIi) there is a next state of the computation of the form <G-{~B},H,E$\cup$Ej,QI$\cup$QIj>.

Logical semantics

The correctness result given by Chan is:

R6.2.1 *Correctness of SLDCNF*

If every branch of the search tree for program P and goal G is finite, and N1,..,Nk are the set of normalised answers for all the success terminating branches,

comp(P) $\models R^T{}_G = R^T{}_{N1} \cup \ldots \cup R^T{}_{Nk}$, T the tuple of variables of G.

Chan gives no completeness results.

Przymusinski (1989) gives a more abstract account of SLDCNF, relating it to the more general problem of reducing a query to an equivalent *equality* formula. He gives normal form results for equality formulas (any well formed formula that contains only {*true*, *false*,=, ≠} that generalise the normal form results of Chan (1988). Przymusinski also gives completeness results for certain restricted classes of programs.

SLDCNF with qualified answers

(Chan 1989) extends SLDNF by allowing partial development of the branches of the search tree for negated calls. The branches of the search tree for a negated call can end in states that have a non-empty G component. The answer returned by such a terminal state is then a qualified normal answer, the qualification being the set of calls in G. This allows finite sets of answers to be returned when the full SLDCNF search tree for the negated call would be infinite. In effect, the extended scheme allows coroutining between the evaluation of positive and negative calls, and generalises the similar coroutining between the manin and auxiliary computations that we discussed for SLDNF.

Other approaches to constructive negation

(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 Logic Programming, the CLP scheme - Jaffar and Lassez

Jaffar 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 requirement for *E* becomes a *satisfaction completeness* requirement for *C*. The following is a slight generalization of the variant of the CLP scheme given by Maher (1987), which better fits the framework of this paper. SLD, AGLD, GLDE, RTS and IRTS are special cases of this CLP scheme.

The theory *C* is a theory for a set of reserved predicates *R* which, remember, always includes *{false, true, =}*.

A *primitive* constraint is an atom with a predicate from *R*. An *allowed* constraint is some subset of all the formulas that can be constructed from the primitive constraints. It minimally contains all equations and is closed under conjunction.

For SLD, *R={false, true, =}*, *C* is HET and only conjunctions of equations are allowed constraints.

For IRTS, *R={false, true, =, ≠}*, *C* is ∃≠RTET, 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

$\mathcal{C} \models \exists C$ or $\mathcal{C} \models \sim\exists C$

This is the generalization of properties R2.2, R2.3 of HET. ∃≠RTET has this property for the allowed constraints of IRTS.

Syntax

CLP programs comprise implications of the form A ← C , G where C is an allowed constraint, A is an atom with a predicate r∈*P* and G is a conjunction of program calls - calls with predicates from *P*. Both C and G can be empty.

A goal is a conjunction of program calls. The lack of an allowed constraint in the goal is no handicap. We can instead have an extra 0-adic atom A in the goal defined by a single rule A ← C added to the program.

Operational semantics

A state of the computation is a triple <G,S,C> where G is a multiset of program calls, S is a satisfiable multiset of allowed constraints for theory C or *false*, and C is a multiset of constraints.

The initial state for goal G is <G,{},{}>. A success termination state is of the form <{},S,{}> with S (or some normalised form of the subset of S *related* to goal G) the computed answer. A fail state has *false* for S.

The computation rule selects some multisubset G'={B1,..,Bk} of calls from G, and some multisubset of C' of the constraints in C. Let

$A1 \leftarrow C1,G1$ .... $Ak \leftarrow Ck,Gk$

be k clauses for the predicates of the selected atoms (as usual with variables renamed to avoid clashes). For each different set of such clauses, there is a next state of the computation of the form

$<G-G' \cup G1 \cup ... \cup Gk,\ S',\ C-C' \cup C1 \cup ... \cup Ck \cup \{B1=A1,..,Bk=Ak\}>$

where

S' is $S \cup C'$ if $C \models \exists S,C'$, false if $C \models \sim\exists S,C'$.

Logical semantics

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

R7.1.1 *Soundness*

If G, with variables T, has a computed answer C then

$C, P \models R^T{}_C \subseteq R^T{}_G,\ R^T{}_C \neq \text{false}$

R7.1.1 *Strong completeness* (the analogue of R3.3.3 for SLD)

If $C, P \models R^T{}_C \subseteq R^T{}_G, R^T{}_C \neq$false for some constraint C then, for any computation rule, G has a k successful derivations with final constraints C1,..,Ck such that

$C \models R^T{}_C \subseteq R^T{}_{C1} \cup ... \cup R^T{}_{Ck}$.

So, for a general constraint theory, we cannot expect to cover C with just one computed answer as we can with the HET schemes. As an example of this result, Maher gives the program

p(a,b)

$p(X,b) \leftarrow X \neq a$

where $R=\{true, false, =, \neq\}$ and C is HET.

For given constraint C={Y=b} and goal G={p(X,Y)}, we do have the precondition

$P,HET \models R^{<X,Y>}{}_{\{Y=b\}} \subseteq R^{<X,Y>}{}_{p(X,Y)} = \{<X,Y>:X=a,Y=b \vee X\neq a,Y=b\}$

However, no single computed answer covers C. We need both the computable constraint answers {Y=b,X=a} and {Y=b,X≠a} to cover it. We do have

$HET \models R^{<X,Y>}{}_{\{Y=b\}} \subseteq R^{<X,Y>}{}_{\{X=a,Y=b\}} \cup R^{<X,Y>}{}_{\{X\neq a,Y=b\}}$

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),$C \models \sim\exists G$ iff for a fair computation rule every branch of the computation tree for G terminates in failure.

This is the generalization of results R3.3.2, R3.3.5 for SLDNF. If *C* includes the normal axioms for equality, the anologue of R3.3.3 should hold (I have not checked the details):

R7.1.4 For goal G, if every branch of the computation tree terminates and C1,....,Ck are all the answers for the success branches, then
completed(P), $C \models R^T{}_C = R^T{}_{C1} \cup ... \cup R^T{}_{Ck}$

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(1987) also extends the above scheme to incorporate the notion of committed choice with the concept of suspension until some guard subset GC of the constraints of a clause is *valid* for the current environment of satisfied constraints S, or is the only satisfiable constraint in the context S of the alternative clauses. GC with variables T is *valid* iff $C \models R^T{}_{GC} \subseteq R^T{}_S$. The unifications with the input argument terms and the evaluation of the guard calls in Parlog and GHC meet this validity condition. The satisfiability condition is similar to the computation rule of P-Prolog.

Saraswat (1988, 1989) 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. That is, the constraints introduced by the use of a clause are not immediately checked for consistency with the satisfied constraints S, they are checked *eventually* when selected by the computation rule.

Instances of the scheme

In any implementable 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 computational step requiring an inference from some first order theory. As Jaffar 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 ideally have a *canonical form*, an equivalent simplified representation using a minimal number of constraints. This would be used for presenting answers and, if possible, 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 the SLD and AGLD instances of the CLP scheme, the canonical form for the satisfied constraints in the computation state is an Herbrand assignment.

The canonical form for answers is a substitution.

As with the GLDE scheme, the great strength of the CLP scheme is that it provides a very general framework for extensions and modifications of the unification based SLD. We simply need to ensure that the algorithm that replaces unification in the proposed new SLD derivative scheme has a validating consistent first order theory *C*. That is, when the algorithm is applied to some allowed constraint C the algorithm correctly determines whether or not $C \models \exists C$. We then know that the above logical properties hold for the computed answers. If the algorithm reduces C to a solution form S, we also need to have the property $C \models RT_C = RT_S$. Then all the obove logical propeites of the CLP scheme will apply.

Implementations of CLP

Prolog III (Colmerauer 1987) is an extension of Prolog II where the constraints are equations and inequations over rational tree 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 reducibility to solution form of any allowed constraint.

CLP(*R*) (Jaffar & Michaylov 1987) has equations over Herbrand terms and inequalities and equations of arithmetic expressions over the real numbers has allowed constraints. 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 (Aiba et al 1988) and CHIP (Dincbas et al 1988) are other constraint languages. EQLOG (Goguen & Meseguer 1986) is also a constraint language since it solves arithmetic equations using special purpose algorithms as well as general term equations using *E*-unification.

8 Concluding remarks

We have seen that the unification of SLD is just a special case of a more general operation of checking that a set of allowed constraint formulas C, containing only a reserved set of predicates *R*, is consistent relative to another set of constraint formulas S for some pre-interpretation PI of the reserved predicates. S is already known to be consistent for PI. PI is a model of some constraint theory *C*.

A logic programming computation uses the clauses of a program to incrementally and non-deterministically generate candidate sets of constraint formulas C to check for PI consistency. It does this by reducing a call B to a program defined predicate to a constraint set C and a set G of calls to program defined predicates, which may be empty, using a clause of the program. The answer to a query comprising a set of calls to program defined relations (and optionally a set of constraint formulas) is any PI consistent set of allowed constraint formulas to which the query can be reduced.

At one extreme, the computation can delay the check for consistency of the union of constraint sets C added during the evaluation of the query, until the query has been reduced to a set which contains no call to program defined relation. The cost is a much larger search tree, with more infinite branches.

At the other extreme, as each call is reduced using some clause, it can check that the C added by the use of the clause is consistent with all the other C's added by the reductions of its ancestor calls. This previously checked union of C's is stored, possible in normalised form, as a separate component of the state of the computation.

Unification is special because there are simple algorithms, some parallel, for checking that each new C, which is a set of term equations, is consistent relative to the equations added by the reductions of the ancestor calls for the Herbrand interpretation HI. In addition, there is a normal form, a Herbrand assignment, which is an explicit representation of the solutions of any HI consistent set of equations. The unification algorithms will check consistency of each new set of equations E relative to the union of the previously added equations E', or relative to the normalised Herbrand assignment form of E'.

Parallel schemes allow concurrent adding of sets of constraints C by the concurrent reduction of a set of program calls. Different ways of checking that the concurrently added sets are pairwise consistent, and consistent with all previously added sets, lead to very different programming language options. The most interesting idea in this area is that due to Maher (1987), later refined by Saraswat(1988,1989), to distinguish between an *ask* subset of the newly added constraints and a *tell* subset. This is the generalization of the data flow ideas embedded in the concurrent unification based languages Parlog and GHC. I anticipate that further work on these abstract models, and on parallel constraint checking algorithms, will lead to powerful generalizations of Parlog and GHC.

The extensions of such languages beyond committed choice, to include ideas such as P-Prolog's parallel selection of deterministic calls and suspension of non-deterministic calls, will further enrich the application area of such languages. The Andorra computation model, due to David Warren, in which deadlock for the *select only deterministic calls* computation rule is broken by selecting and non-deterministically exploring alternative evaluation paths for any of the non-deterministic calls, is the key to the extension. It has already been incorporated into Andorra Prolog (Haridi & Brand 1988) and the Pandora (Bahgat and Gregory 1988) extension of Parlog.

Extensions of negation as failure to allow negated calls to return answers necessarily leads us from simple equation solving to constraint satisfaction, as we saw with the SLDCNF scheme. Setting this work in a more general framework, which has already been started by Przymusinski (1989), will pay dividends. What we need are general properties of constraint sets that will allow the mapping of the complement of the union of sets of allowed constraints into a union of a sets of allowed constraints. We then need efficient algorithms for doing this for sets of allowed constraints richer than those of the Chan scheme. Finally, we need to extend the completeness results for the checking role of the negation as failure rule of SLDNF to a scheme that allows for constraint generating negated calls.

The area of the integration of functional and logic programming is becoming more active. The Holldobler (1990) book, already mentioned, is a good introductory reference. Further work in this area should lead to well integrated functional and logic programming languages, such as K-LEAF (Giovanetti et al 1989), perhaps with a constraint logic programming component.

Work has also started on the integration of ideas of object oriented programming with logic programming. (McCabe 1989) defines an extension of SLDNF which deals with inheritance of static objects - objects for which there is no change of state. Other approaches are surveyed in (Davison 1990). (Goguen & Meseguer 1987) is an ambitious attempt to integrate functional, object oriented and logic programming as an extension of EQLOG. A clean integration, preserving the logical semantics as much as possible, will give us logic languages that can use the program structuring ideas of object oriented programming. (McCabe 1989) has simple examples of the usefulness of these program structuring ideas.

References

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

Apt., K.R. (1990) Introduction to Logic Programming, to appear in *Handbook of Theoretical Computer Science*, (ed van Leeuwen, J), North-Holland.

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

Bahgat, R., Gregory, S. (1989) Pandora: Non-deterministic parallel logic programming, *ICLP6*, MIT Press.

Barbuti, R., Martelli, M., (1986), Completeness of SLDNF resolution for a class of logic programs, *ICLP3*, Springer-Verlag LNCS 225.

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

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

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

Cavedon, L., Lloyd, J.W., (1989), A completeness theorem for SLDNF resolution, *JLP 7(3).*

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

Chan, D., (1989), An extension of Constructive Negation and its application to Coroutining, *Proc. NACLP 89*, 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 formalism, Research report, Logic Programming Section, Department of Computing, Imperial College.

Clark, K. L., (1988), Logic Programming Schemes, *Proceedings of FGCS88*, ICOT, Tokyo, Japan.

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

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

Toplas 8(1).

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 (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.

Davison, A. (1990) Design issues for logic based object oriented programming languages, Research Report, Parlog Group, Dept. of Computing, Imperial College, London.

Drabent, W., Martelli, M., Strict completion of logic programs, Research Report R-89-28, Institutionen for Datavetetenskap, Universitetet och Tecniska Hogskolan, Linkoping, Sweden.

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., (1990), Absys: The First Logic Programming Language - A retrospective and a commentary, to appear in *JLP*.

van Emden, M. H., Kowalski, R. A. K., (1976), The semantics of predicate logic as programming language, *JACM*, 23, 4, 733-742.

van Emden, M.H., Lloyd, J.W., (1984), A logical reconstruction of Prolog II, *ICLP2*, Upsalla University.

van Emden, M. H., Keitaro, Y. (1987) Logic programming with equations, *JLP* 4(4).

Fay, M. (1979) First order unification in an equational theory, *4th workshop on automated deduction*, Austin.

Foo, N., Rao, A., Taylor, A., Walker, A., Deduced relevant types and constructive negation, *ICLP5*, MIT Press 1988.

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

Gallier, J. H., Raatz, S. (1986) SLD-resolution for Horn clauses with equality based on E-unification, *Proc. SLP-86*, IEEE.

Giovanetti, E., Levi, G., Moiso, C., Palamidessa, C. (1989) Kernel LEAF: a logic plus functional language, Tech. Report TR-2/89, Dipartimento di informatica, Universita di Pisa.

Goguen,J.A., Meseguer, J. (1986) EQLOG: Equality, types and generic modules for logic programming, in *Logic Programming Functions, Relations and Equations* (ed. Degroot, D. & Lindstrom, G.). Prentice-Hall.
Gregory, S. (1987) *Parallel Logic Programming in Parlog*, Addison-Wesley.

Goguen,J.A., Meseguer, J. (1987) Unifying functional, object-oriented and relational programming with a logical semantics, in *Research directions in Object Oriented Programming*, (ed Shriver, B., Wegner, P.), MIT Press.

Haridi,S., Brand,P. (1988) Andorra Prolog - an integration of Prolog and committed choice languages, *FGCS88*,ICOT

Herbrand, J. (1930), Recherches sur la theorie de la demonstration, These, University of Paris. Also in *From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931,* (ed. van Heijenoort), Harvard University Press, 1967.

Hermenegildo, M., Rossi,F. (1989) On the correctness and efficiency of independent and-parallelism in logic programs, *NACLP89*, MIT Press.

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

Holldobler, S. (1987) Equational logic programming, *Proc. SLP-87*, IEEE.

Holldobler, S. (1988) From paramodulation to narrowing, *ICLP-5*, MIT Press.

Holldobler, S. (1990) *On the foundations of equational logic programming*, LNCS 353, Springer Verlag.

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

Jaffar, 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.

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

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

Jaffar, J., Lassez, J-L., Maher, M. J., (1987) Prolog II as an instance of the logic programming language scheme, *Formal Description of Programming Concepts III*, (ed Wirsing M.), North-Holland.

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

Jouannaud, J-P., Kirchner, C., Kirchner, H. (1983) Incremental construction of unification algorithms in equational theories, *Automata, Languages and Programming*, Barcelona.

Kaplan, S. (1986) Fair conditional term rewriting systems: unification, termination and confluence, *Recent trends in data type specification* (ed Kreowski), 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, E., (1988), The Language FCP(!,:,?), *FGCS 88*, ICOT

Kowalski, R. A., (1974), Predicate logic as a programming language, *IFIP 74*.

Kunen, K., (1987), Answer sets and negation as failure, *ICLP4*, Melbourne, MIT Press.

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

Kunen, K., (1989) Signed data dependencies in logic programs, *JLP 7(3)*.

Lassez, J. L., Maher, M.J. (1984) Closure and fairness in programming logic, Theoretical Computer Science.

Lassez, J. L., Maher, M.J., Marriot, K.L., (1988), Unification revisited, in (Minker 1988), Morgan Kaufmann.

Lloyd, J.W. (1984), *Foundations of Logic programming*, Springer-Verlag.

Lloyd, J.W., Topor, R.W., (1984), Making Prolog more expressive, *JLP 1(3)*.

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

Maher, M.J., (1987), Logic semantics of a class of committed choice programs, in *ICLP 4*, MIT Press.

Maier, D., Warren, D. S., (1988), *Computing with Logic*, Benjamin/Cummins.

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

Martelli, A.,Moiso, C., Rossi, G. F. (1986) An algorithm for unification in equational theories, *Proc. SLP-86*, IEEE.

McCabe, F. G. (1989) Logic and objects, PhD thesis, Dept. of Computing, Imperial College, London.

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.

Paterson, P. A., Staples, J., (1988), A General Theory of Unification and Solution of Constraints, Technical Report No. 90, Dept. of Computer Science, University of Queensland, Brisbane.

Plotkin, G. D., (1972) Building in equational theories, *Machine Intelligence 7*, (ed Meltzer, B., Michie, D.), Halsted Press.

Przymusinski, T. (1989) On Constructive Negation in Logic Programming, Proc. *NACLP 89*, 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., (1965), A machine oriented logic based on the resolution principle, *JACM 12 (1)*.

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

Sato, T., (1987), On consistency of first order logic programs, Tec. Report 87-12, Electrotechnical Lab., Ibaraki, Japan.

Saraswat, V. J., (1987), The concurrent logic programming language cp: definition and operational semantics, *POPL87*, ACM.

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

Saraswat, V. J., (1989) Concurrent Constraint programming languages; PhD thesis, Carnegie-Mellon University.

Siekmann, J. H. (1984) Universal unification, *Proceedings CADE-7,* LNCS 170, Springer-Verlag.

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

Shapiro, E. (1989) The family of concurrent logic programming languages, Research report CS89-08, Weizmann Institute of Science, 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., Furukawa, 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.

Taylor, S. (1989) *Parallel Logic programming Techniques*, Prentice-Hall.

Thom, 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.

Yamamoto, A. (1987) A theoretical combination of SLD-resolution and narrowing, *ICLP4*, MIT Press.

Yang, R., Aiso,H. (1986) P-Prolog: parallel language based on exclusive relation, *ICLP3*, LNCS 225, Springer-Verlag.