Extending Prolog with Type Inheritance and Arithmetic.

by

Chandan S. Chitale

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Science and Applications

APPROVED:

J. W. Roach, Chairman

L. T. Watson  J. P. Bixler

May, 1989
Blacksburg, Virginia
Extending Prolog with Type Inheritance and Arithmetic.

by

Chandan S. Chitale

J. W. Roach, Chairman

Computer Science and Applications

(ABSTRACT)

Prolog is a logic programming language based on first order logic. It uses resolution as a rule of inference, and unification is the heart of resolution. The unification algorithm is a syntactic process and hence attaches no meaning to function and predicate symbols. We incorporate arithmetic into unification by simultaneously solving linear equations that are created during the unification of partially instantiated numeric expressions. Prolog operates on the Herbrand universe, which is a single unstructured domain. In case of large structured domains, the number of resolution steps required for inference is large. We have incorporated type inheritance into Prolog to exploit large structured domains. Types are subuniverses corresponding to sets of objects. The ‘subset of’ relation between types induces a hierarchy on the universe. Using the property of inheritance it is possible to obtain shorter proofs in inference. We used the constraint satisfaction model and the hierarchical constraint satisfaction concept to incorporate these extensions to Prolog. Thus, we succeeded in obtaining a logic programming language with arithmetic and type inheritance. This implementation extends standard Prolog and can be directly added to the WAM concept.
Acknowledgements

I would like to thank Dr. Roach for his guidance throughout this work and also Dr. Watson for helping me with all the Linear Algebra concepts I needed to know. I would also like to thank Dr. Bixler for serving on my committee.

I must thank for all his help, the hours spent discussing ideas and explaining concepts to me. I want to thank for hearing me out, correcting me and even cheering me up.

My husband I thank for endless support, tolerance to my moods, and constant calls to boost my morale. My parents and parents-in-law, I thank for their moral support.
Table of Contents

1. Introduction. ......................................................... 1
   The Objective. ........................................................ 1
   The Achievement. ..................................................... 2
   Outline. .................................................................... 4

2. Related Work. ........................................................ 5

3. The General Problem. ................................................. 12
   Prolog. .................................................................... 12
   Incorporating Arithmetic into Unification. ......................... 15
   Inheritance. ................................................................ 15
   Constraint Satisfaction. .............................................. 16
   Hierarchical Constraint Satisfaction. .............................. 18
   Object-Oriented Paradigm. ........................................... 20
   The Problem. ........................................................ 21

4. The Constraint Satisfaction Approach to Unification. .............. 23
   Objects. ............................................................ 23
   Functor objects. ...................................................... 23
   Interpreted Functors. ................................................. 24
   Compatible Functors. ............................................... 24
   Terms. ............................................................. 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Structure.</td>
<td>54</td>
</tr>
<tr>
<td>Closed Type Structure.</td>
<td>54</td>
</tr>
<tr>
<td>Complete Type Structure.</td>
<td>54</td>
</tr>
<tr>
<td>Typed Terms.</td>
<td>56</td>
</tr>
<tr>
<td>The Mapping $M$ extended to Typed Terms.</td>
<td>57</td>
</tr>
<tr>
<td>Partially Typed Terms (PTT).</td>
<td>58</td>
</tr>
<tr>
<td>Unification of Partially Typed Terms.</td>
<td>58</td>
</tr>
<tr>
<td>Unification of typed terms.</td>
<td>58</td>
</tr>
<tr>
<td>Correctness of the implementation.</td>
<td>59</td>
</tr>
<tr>
<td>Declared Type Structure.</td>
<td>59</td>
</tr>
<tr>
<td>8. Time and Space Complexity Analysis.</td>
<td>68</td>
</tr>
<tr>
<td>9. Results.</td>
<td>72</td>
</tr>
<tr>
<td>Incorporating Arithmetic.</td>
<td>72</td>
</tr>
<tr>
<td>Type Inheritance.</td>
<td>74</td>
</tr>
<tr>
<td>10. Conclusions and Future Work.</td>
<td>82</td>
</tr>
<tr>
<td>Comparison with LOGIN.</td>
<td>82</td>
</tr>
<tr>
<td>Bibliography</td>
<td>85</td>
</tr>
<tr>
<td>Appendix A. User's Manual.</td>
<td>88</td>
</tr>
<tr>
<td>Appendix B. Extending the Incorporation of Arithmetic into Unification.</td>
<td>93</td>
</tr>
<tr>
<td>Appendix C. Changes to the WAM to incorporate Type Inheritance.</td>
<td>94</td>
</tr>
</tbody>
</table>

Table of Contents
Get Instructions. ...................................................... 95
Put Instructions. ...................................................... 96
Unify Instructions. ..................................................... 96
List of Illustrations

Figure 1. Data Structures for Incorporating Arithmetic to Unification. .............. 40
Figure 2. Data Structures for Adding Types with Inheritance to Unification. .... 46
Figure 3. Type Inheritance: Example 1. ......................................................... 79
Figure 4. Type Inheritance: Example 2. ......................................................... 80
Figure 5. Type Inheritance: Example 3. ......................................................... 81
1. Introduction.

In this thesis we enhance the capabilities of Prolog by including arithmetic function evaluation and the concept of inheritance by using constraint satisfaction. We introduce the objective and the achievement here and also give an outline of the following chapters.

The Objective.

Prolog is a logic programming language based on Horn clause logic, a subset of first order logic. It uses resolution as a rule of inference. The heart of resolution is unification.

The Unification algorithm used, is a purely syntactic process. There is no semantic understanding of predicates and function symbols involved. Two functions will unify only when the arguments are syntactically equal. Hence the meaning of the function symbols is not exploited.

Consider the expressions

\[ [+ \ ?y \ 6] \text{ and } [+ \ 2 \ 7]. \]

These will not unify as \(?y\) and 2, and 6 and 7 are not equal.

Consider the expressions

\[ [- \ ?x \ 5] \text{ and } [+ \ ?y \ [* \ 2 \ ?x]]. \]
These will not unify as the function symbol itself does not unify.

Prolog operates on a single unstructured domain, the Herbrand Universe. Hence programs based on structured domains take many resolution steps. Consider the following example in standard Prolog.

\[(\text{animal} ?x) \text{ if } (\text{bird} ?x))\]
\[(\text{animal} ?x) \text{ if } (\text{mammal} ?x))\]
\[(\text{mammal} ?x) \text{ if } (\text{whale} ?x))\]
\[(\text{whale} \text{ mobyduck1})]\]
\[
\ldots\]
\[(\text{whale} \text{ mobyduck20})\]
\[
(\text{breathes} ?y \text{ air} ) \text{ if } (\text{animal} ?y))\]

Consider the query (\text{breathes mobyduck20 air}).

Solving this query takes 4 resolution steps and several backtracking steps to result in the goal being true.

**The Achievement.**

To overcome the purely syntactic nature of unification we incorporated arithmetic into unification. This was implemented by adding a linear numerical constraint solving module to the unification procedure. Consider the previous examples.

The first example will result in the binding \(?y = 3\), and the second example will result in an equation of the form, \(?x + ?y = -5\).
Evaluation of functions when all arguments are instantiated, although not a part of pure Prolog, is a part of all implementations of Prolog. We will be addressing the evaluation of functions, where not all arguments are instantiated.

We have also modified unification to include a type inheritance mechanism. Thus the domain of discourse can be subdivided into several types. These subuniverses could be disjoint, or intersecting, or subsets of each other. Using the syntax of our system, we declare the above information as follows.

(bird < animal)
(mammal < animal)
(whale < mammal)
(mobydick1 << whale))
.....
(mobydick20 << whale))
((breathes X: animal air))

Now, the same query will result in a true value in a single resolution step, with no backtracking because the inheritance information is represented in the machine and is used during unification. Untyped examples in standard Prolog can still be handled as usual.

The inheritance mechanism is an important feature of the object oriented paradigm. Thus by incorporating inheritance we are adding a new functionality to Prolog. Also incorporating function evaluation into unification increases the expressive power of Prolog.

1. Introduction.
Outline.

In Chapter 2 we review the relevant work in this field. In Chapter 3 we discuss the problem and what we want to achieve. In Chapter 4 we give an algorithm which draws the two ideas together. In Chapter 5 we discuss the incorporation of numerical constraint solving and in Chapter 6 the implementation of type inheritance. In Chapter 7, we will formalize the theory behind the type inheritance. In Chapter 8 we dwell on the complexity of both implementations. In Chapter 9, we give the results of the implementation and in Chapter 10 the conclusions and future work.
2. Related Work.

Prolog III.

Prolog III is an extension of Prolog II [GIAN 86]. List processing, complete processing of Boolean algebra and number processing have been integrated at the unification level of Prolog [COLM 87]. This reshaping consists of replacing the unification concept by the concept of constraint resolution in a domain with appropriate operations and relations defined on the domain.

Prolog III solves systems of simultaneous arithmetic and boolean constraints. The heart of the interpreter is an algorithm to solve systems of constraints. The algorithm first determines whether the constraints are consistent and then simplifies them to obtain the solution or an empty system denoted by { }. The constraint solver has a module to process Boolean algebra and one to process numerical constraints.

A term is represented as a tree consisting of variables, constants and operations. The set of facts that can be deduced from a program is like an enormous (usually infinite) database. The execution of a program is a search through a fraction of the database. The database is stored as a set of rules.

The core of the interpreter solves most of the equations and the inequality constraints. It calls on two sub-modules to do specific processing. One module is for
processing Boolean algebra. No details are given about this module. Arithmetic processing is implemented using linear programming techniques. There is a ‘subtle process’ to take care of degeneracies, but this has not been described. Constraints of the \( \neq \) type have also been handled, but again the method has not been elaborated.

Prolog III does not handle incremental addition of constraints. Either the newly added constraint variables are disjoint or some method of constraint propagation needs to be introduced.

**Constraint Logic Programming.**

CLP(R) is an experimental implementation of the Constraint Logic Programming paradigm. CLP uses the resolution principle, like logic programming, but the concept of syntactic unification in the Herbrand Universe is replaced by constraint satisfaction in the domain of the application. Answers to a goal are determined by solvability of the corresponding set of constraints and not in terms of finding a specific value as the solution. Therefore, if there is no solution for a set of constraints but the system is consistent, the solution is the set of simplified constraints itself. The sets of linear constraints are solved using a modified simplex method. The processing of non-linear constraints is delayed until a sufficient number of variables are instantiated and the constraints become linear. Incremental addition of constraints is possible, but exactly how this addition is implemented is not clearly explained.

The CLP(R) Interpreter [LASS 87] uses a Prolog-like engine, a constraint solver and an interface module. The engine has an inference mechanism by which it determines the type of constraints and solves simple constraints. The interface module evaluates
arithmetic expressions and puts them in a canonical form. The constraint solver solves constraints and also provides the delay mechanism for non-linear constraints.

Constrained Unification.

[ROAC 89] discusses the replacement of unification by a constraint solver. The generalization of unification into a constraint satisfaction algorithm allows the incorporation of function evaluation into unification.

This system was implemented in Prolog as a meta interpreter. The types of numerical constraints that were considered were, ≤, ≥, =, ≠, and ∈. Set membership was defined on finite sets that do not allow sets as their elements. Inequalities were handled using linear programming techniques. Negation was also implemented by delaying solving the goal until all the arguments were instantiated. An incremental consistency checking algorithm was also used.

Many-Sorted Unification.

In [WALT 88], the universe is divided into subsets or sorts and is defined as a potentially infinite hierarchy of sorts. Many-sorted unification (Σ-Unification) is discussed as being equivalent to solving an equation in the corresponding heterogeneous many-sorted algebra rather than in a homogeneous algebra (Robinson’s unification).

The conditions imposed on the structure of the sort hierarchy required to classify the Σ-unification completely are considered. Also, cases where complete and minimal sets of Σ-unifiers do not always exist are discussed. The Robinson Unification Theorem requires the existence of most general unifiers which exist only if the sort
structure is a forest structure. Otherwise, 'auxiliary variables' have to be added to obtain complete and minimal sets of Σ-unifiers. The Σ-unification algorithm returns a set of substitutions, unlike normal unification. To obtain a singleton set of complete and minimal Σ-unifiers, the structure needs to be a meet semi-lattice (i.e., every pair of sorts has a greatest lower bound). It is possible to embed any sort hierarchy in a meet semi-lattice by inventing additional sorts.

**Extended Prolog for Order-Sorted Resolution.**

EPOS [HUBE 87] is an elaboration of the Prolog language based on order-sorted resolution. It supports data abstraction and an inheritance mechanism. Linear Resolution with Selection for Definite Clauses or SLD resolution is extended to order-sorted resolution by replacing the Robinson-unification algorithm by order-sorted resolution. EPOS implements the many-sorted unification idea, put forth by [WALT 88].

EPOS supports both one-sorted and order-sorted resolution. A "mixed" execution is not possible. The execution mode has to be determined at startup.

EPOS queries and programs are rather insensitive with reference to sub-goal (and clause) ordering, unlike Prolog. Multiple type declarations (overloading) for a single function are not allowed. The algorithm assumes that a greatest lower bound exists for every pair of sort symbols in the system.

**LOGIN.**

[AITK 86] uses a paradigm of unification that allows the separation of multiple inheritance from the logical inference machinery of Prolog.

2. Related Work.
A declaration of inheritance information is called a ‘signature’. The type signature $\Sigma$ is a partially ordered set of symbols that contains two special symbols, a greatest element ($T$) and a least element ($\perp$). Type symbols denote sets of objects, and the partial order on $\Sigma$ denotes set inclusion. Unification is defined as the process of computing the greatest lower bound (glb) of two symbols relative to the set inclusion ordering. Generalization is defined as the process of computing the least upper bound (lub) of two symbols relative to the set inclusion ordering.

A syntactic representation of a structured type is used, called the $\psi$ term. A $\psi$ term consists of a root symbol, attribute labels with associated sub-$\psi$ terms and coreference constraints among paths of labels. Conventional first-order terms are a particular instance of the $\psi$ term. $\psi$ terms are treated like records with attributes as fields. By introducing attributes to $\psi$ terms, it is possible to have a variable number of arguments and an arbitrary ordering of arguments. Attribute labelling adds coreference constraint variables to LOGIN.

A well formed term (wft) is defined as a $\psi$ term, where all occurrences of tag symbols are of the same structured type. Thus terms represent types and therefore the concept of a term being a subtype of a term exists. In fact, the wft ordering extends the semi-lattice from the signature to the wft’s. Thus the unification of wft’s is the wft that represents the GLB of the two wfts.

The standard unification algorithm is replaced by the $\psi$ term unification algorithm. This algorithm computes the $\psi$ term which is the greatest lower bound of the two given $\psi$ terms. The tags for attribute labels are used to build coreference classes that have class representatives. All tags in the unified $\psi$ term are replaced by their class.

2. Related Work.
representatives, thus resolving coreference. \( \Sigma \) unification returns \( \perp \) (if the \( \psi \) terms cannot unify) or the \( \psi \) term built after resolving coreference.

The sorts structure needs to be a lower semi-lattice for this algorithm to work, i.e., \( \text{glb}'s \) must exist for all pairs of sort symbols in the sort structure. In case the structure is not a lower semi-lattice, it is embedded into the least structure that contains it by using the restricted powerset \( 2^{\omega} \) of the partially-ordered set \( (S, \leq) \) of sort symbols. The set \( 2^{\omega} \) consists of the nonempty finite subsets of pairwise incomparable elements of \( S \).

In the implementation described, the main emphasis is on the unification algorithm. In this algorithm, they discuss the effects of backtracking. As coreference classes need to be 'unmerged' they did not physically merge them, but used dereferencing. The \( \psi \) terms store their own backtracking information. There is only one field for the type constructor symbol, hence the outcome of backtracking after a series of unifications is not indicated.

The \( \text{glb} \) operation is the most frequently used operation during unification, but nothing is mentioned as to how it is achieved. The \( \text{glb} \) must represent the true intersection of the terms being unified, but a check for the correctness of the user's structure is not mentioned.

Although LOGIN can handle programs in untyped standard Prolog, the \( \psi \) term concept has moved LOGIN away from standard Prolog syntax and semantics that are incorporated in the WAM [WARR 83].

**Object-Oriented Programming in Prolog.**

2. Related Work.
[ZANI 84] is an extension of UNSW Prolog that defines a new syntax to declare objects, methods and message passing, the three main ideas behind object-oriented programming. The object-oriented syntax uses infix operators. It is possible to define objects with their associated set of methods. The application of methods to objects is message passing. Objects are related to each other by inheritance relations. Methods are basically predicates defined for a particular object. Objects inherit methods from their ancestors. Methods are statically bound to objects. When a message specifies the passing of a method M to an object O, the interpreter first attempts to unify M with the methods associated with O, otherwise it attempts to unify M with the methods associated with the ancestors of O.
3. The General Problem.

In this chapter we discuss Prolog, Robinson’s Unification algorithm and the limitations of both. Then we introduce the ideas of incorporating arithmetic and inheritance into unification to enhance the capabilities of Prolog.

**Prolog.**

Prolog is a logic programming language used to perform automatic deductions based on a refinement of resolution theorem proving for first order logic. A Prolog program consists of Horn clauses and uses resolution to make inferences. We define a few concepts in logic [LLOY 84] to make the discussion clearer.

**Terms:**

A term is defined inductively as follows:

1. A variable is a term.
2. A constant is a term.
3. If f is an n-ary function symbol and \( t_1, \ldots, t_n \) are terms, then \([f \ t_1,\ldots,t_n]\) is a term.

A term with no variables is called a **ground term**.
**Atomic Formula**

A well-formed formula (WFF) \((p, t_1, \ldots, t_n)\) in which \(p\) is an \(n\)-ary predicate and \(t_1, \ldots, t_n\) are terms is called an atomic formula.

**Literal**

An atomic formula or the negation of an atomic formula is called a literal. A positive literal is an atomic formula, and a negative literal is the negation of an atomic formula.

**Clause.**

A clause is a formula of the form

\[ \forall x_1, \ldots, a_i x_n (L_1 \lor \ldots \lor L_m), \]

where each \(L_i\) is a literal and \(x_1, \ldots, x_n\) are variables that occur in these literals.

**Horn Clause**

A clause that contains exactly one positive literal and zero or more negative literals is called a Horn clause. The positive literal is called the head of the clause and the negative literals together are called the body of the clause. Any set of sentences in first order logic can be re-expressed as a set of Horn clauses. A Horn clause with only a head is called a fact while all other Horn clauses are called rules.

**Goal Clause**

A clause consisting of only negative literals (no head) is called a goal clause.

**Substitution**
A substitution is a finite set of the form \( \{v_1/t_1, \ldots, v_n/t_n\} \), where each \( v_i \) is a distinct variable, and each \( t_i \) is a term distinct from \( v_i \). A substitution is called a ground substitution if all of the \( t_i \) are ground terms.

**Unification.**

The process of applying a substitution to two terms such that both terms become syntactically equal is called unification. We will refer to this as Robinson’s unification or normal unification.

Prolog is a resolution based theorem prover. Resolution is the process of starting with the negation of the semantic goal and arriving at a contradiction using rules of inference from first order logic. A Prolog program consists of rules and facts. A query in Prolog is a goal clause. The Prolog runtime machine unifies the goal clause with the head of a horn clause. The literals in the body of the Horn clause then become the sub-goals of the query. These in turn are unified with other rules and facts. If there is no match, then the sub-goal fails. This process ends when each sub-goal matches a fact.

Thus, unification is the heart of resolution. Unification is purely syntactic in nature. Therefore, the semantic meaning of functions and predicates is ignored. For example, the following expressions would not unify:

\[
(+ 4 4) \text{ and } (+ 5 3)
\]
\[
(+ 6 X) \text{ and } (+ 4 Y)
\]

where \( X \) and \( Y \) are variables.

3. The General Problem.
To understand the semantic nature of functions and predicates the ‘meaning’ of these must be built into the unification process. This could be possible by adding the axioms that describe the function or predicate to the initial set of clauses. This might cause the generation of many useless resolvents. Another approach is to build special procedures that handle this into the unification procedure.

Prolog operates in the Herbrand universe. This consists of the set of all ground terms, which can be formed out of the constants and functions appearing in the Prolog program. This is a single unstructured universe. To instantiate a variable in the program, the search space corresponds to the entire universe.

**Incorporating Arithmetic into Unification.**

Incorporating arithmetic into unification involves understanding the semantics of arithmetic functions and hence being able to evaluate them. We are interested in arithmetic that is defined over the domain of real numbers and integers. We know that these are infinite spaces, and hence to find solutions by exhaustively searching is quite infeasible. We know that there are methods of solving arithmetic goals simultaneously and would prefer to use them.

**Inheritance.**

Large domains are often structured, and groups of elements in the domain tend to behave similarly. In such cases, it makes sense to divide the domain of discourse...
into sets called subuniverses. These subuniverses are partially ordered on the ‘sub-
set of’ relation, which induces a hierarchy on the sets. This domain graph is not a
strict tree and is in fact a directed acyclic graph. Since a subuniverse consists of
objects that behave similarly, properties attributed to the subuniverse are attributed
to all elements of it. Properties of a subuniverse are inherited by all subsets of it.

**Constraint Satisfaction.**

Constraint satisfaction is a model of computation. It is the process of satisfying con-
straints locally and then propagating the results globally. A constraint satisfaction
problem consists of variables, each with an associated domain and a set of con-
straining relations that involve a subset of these variables. All possible tuples that
satisfy the relations are solutions to this problem. To obtain a solution, a step-wise
refinement process is implemented until the solution space gets sufficiently pruned
and a set of solutions is found that satisfy all the constraints. This iterative procedure
is called constraint propagation. Constraint solvers excel in quantitative reasoning
as well as symbolic inferencing.

Constraint satisfaction problems are characterized by:

1. A finite set of variables K.

2. A finite non-empty set of values V.

3. Associated with each variable i, a set of possible values $D_i \subseteq V$.  

3. The General Problem.
4. Associated with each (unordered) pair of variables \( i, j \) a constraint relation \( R_{ij} \subseteq D_i \times D_j \).

5. The set of all possible assignments

\[ \{A | A \in D_1 \times D_2 \times \ldots \times D_k \} . \]

6. Solution criterion:

An assignment \( A \) is a solution iff

\[ \forall i, j \leq k, \text{ if } i \neq j \text{ then} \]

\[ (A(x_i), A(x_j)) \in R_{ij}, \text{ where } A(x_i) \text{ is the value assigned to variable } x_i \text{ in the assignment } A. \]

The constraint system can be represented as a graph \( G \), where the vertices represent the variables in the system. There is an arc between the nodes \( v_i \) and \( v_j \) if the relation \( R_{ij}(x_i, x_j) \) holds for some \( x_i, \ t_j \). \( \Delta_i \) represents the dynamic value of the currently permissible domain of variable \( x_i \). The set of domains \( \{\Delta_i\} \) have to be initialized to \( D_i \).

The core of constraint satisfaction systems is constraint propagation. In this method, first we refine the domain of a variable participating in a constraint by deleting all values for which we cannot find values for all other variables in that constraint from their respective domains.

Constraint propagation can be described using the following algorithms [MACK 77].

\[
\begin{align*}
\text{procedure REVISE-A ((i,j))} \\
\text{begin} \\
\Delta \leftarrow \{x|(x \in \Delta_i) \wedge [(\exists y)(y \in \Delta_j) \wedge R_{ij}(x,y)]\} \\
\text{DELETE} \leftarrow (\Delta \subset \Delta_i) \\
\text{if DELETE then } \Delta \leftarrow \Delta \\
\text{return DELETE}
\end{align*}
\]
end.
{" main *\}
begin
Q ←\{ (i,j) | (i,j) ∈ arcs(G) i ≠ j\}
while Q not empty do
begin
select and delete any arc (k,m) from Q
if REVISE-A((k,m)) then Q ← Q ∪ \{ (i,k) | (i,k) ∈ arcs(G), i ≠ k, i ≠ m\}
end
end.

Thus, this program repeatedly applies the constraint to the domains of the variables and prunes them. The algorithm terminates when the REVISE-A operation produces no more change in the domains of the variables and all the arcs are processed.

Hierarchical Constraint Satisfaction.

Hierarchical Constraint Satisfaction (HCS) is a method of handling constraint satisfaction problems where the variables have large domains by exploiting their internal structure. In fact, for many real world problems the domain elements cluster together into sets with common properties and relations. This structure can be represented as a hierarchy and is partially ordered on the 'subset of' relation. The expectation is that the domains are structured so that the elements of a set frequently share consistency properties permitting them to be retained or eliminated as a unit. Thus, if some elements of a set satisfy a constraint, but not all, the subsets of the set are considered. In this way, if no elements of a set can satisfy the constraint the whole set can be discarded. Thus, structuring the domain helps in considering sets of elements all at a time and hence helps in pruning the search space more quickly.

It is possible to extend constraint satisfaction to HCS.

3. The General Problem.
The domain $D_i$ associated with each variable $i$, can be interpreted as a hierarchy of subdomains. According to [MACK 85] they assume that the domain graph is a singly rooted strict tree and each subset has one superset. Also each non-singleton set consists of two mutually exclusive and exhaustive subsets. Each domain element is a singleton set at the bottom of the hierarchy. The subdomains of $D_i$ are denoted by $\{D_{ij}\}$, where $q$ stands for the level of the domain tree and $s$ the subdomain at that level.

The constraint propagation algorithm is the same, but we replace the REVISE-A by the REVISE-HAC taken from [MACK 85].

```pro\noun{gramme} REVISE-HAC ((i,j));
begin
DELETE ← false;
Q₁ ← Δᵢ
Δᵢ ← φ
while Q₁ not empty do
begin
select and delete an element $D_{ij}$ from Q₁
Q₁ ← Δᵢ
FOUND ← false
while Q₂ not empty and not FOUND do
begin
select and delete an element $D_{ij}$ from Q₂
if $\forall$ elements of $D_{ij}$ exists an element of $D_{ij}$ that satisfies $R_{ij}$ then
begin
Δᵢ←Δᵢ∪{D_{ij}}
FOUND ← true
end
end
if not FOUND then
begin
DELETE ← true
if q > 0 then
begin
Q₂←Δᵢ
while Q₂ not empty and not FOUND do
begin
select and delete an element $D_{ij}$ from Q₂
if for some element of $D_{ij}$ $\exists$ an element of $D_{ij}$ that satisfies $R_{ij}$ then
begin
Q₁←Q₁∪{subsets of $D_{ij}$}
end
end
end
end
end
```

3. The General Problem.
It has been claimed that these algorithms are correct and do terminate.

**Object-Oriented Paradigm.**

The domain of discourse consists of 'objects', each with its own set of procedures called 'methods'. An important feature of this paradigm is the inheritance network whereby an object is declared to be a specialization of other objects, therefore inheriting their attributes and methods. An action is specified by passing a 'message' to an object requesting the execution of one of its methods. Data encapsulation is also a key feature of this paradigm.

Prolog naturally supports data encapsulation as there are only local variables in each clause. By the addition of types to Prolog, a type can be considered to be an object. Clauses that are defined with heads of the form of (P ?x: type1 ?y) where P is a predicate, and type1 is an object and ?y corresponds to the arguments, can be viewed as methods for the object type1. Thus the addition of types and inheritance of types to Prolog links logic programming with the object oriented paradigm.
The Problem.

Given Robinson's unification, arithmetic and inheritance as three points of a triangle, we wish to draw these three elements together to obtain a new kind of logic programming language.

Constraint satisfaction is a model of computation. Unification can be viewed as a constraint satisfaction problem because we are solving simultaneous systems of equations [COLM 84] [DAVI 87]. Unification is more efficient for symbolic inferencing while constraint satisfaction strongly supports both quantitative and symbolic inferencing. Hence replacing unification by constraint satisfaction allows us to combine symbolic unification with numerical function evaluation.

Consider the concept of inheritance. We wish to have structured domains so as to cut down the search space for instantiations, and we need to combine inheritance with unification using constraint satisfaction. The domain is divided into subuniverses that we will refer to as types. Thus each type represents a set of elements from the universe. Consider the following type structure:

1. The structure has two special types 'top' and 'bottom', where top represents the set of all objects in the universe and bottom, the empty set.

2. The domain is structured in such a way that the subsets of a set are exhaustive and mutually exclusive. Each set consists of only two subsets and all objects of the domain are singleton sets which are supersets of bottom.
Unification is the process of satisfying an equality constraint on a pair of variables. In this structured domain, we define variables to belong to a particular domain or type. We wish to unify the two variables \( x \) and \( y \), belonging to types \( D_x \) and \( D_y \), respectively. Let us apply the HCS algorithm to this problem.

- We begin with \( \Delta_x \leftarrow D_x \) and \( \Delta_y \leftarrow D_y \).

- \( \Delta \leftarrow \) all elements \( a_x \in \Delta_x \) for which \( \exists \) an element \( a_y \in \Delta_y \), such that \( a_x = a_y \).

- The algorithm will terminate when we reach a stage where \( \Delta \) contains only sets that contain nothing but intersecting elements. Thus a union of the sets will result in the set of all intersecting elements of \( D_x \) and \( D_y \). This union is the maximal intersection set of elements.

Thus we see that HCS applied to unification defined on a typed universe leads to the idea of intersection of domains. We use the HCS concept to draw unification and inheritance together.

In the next chapter we discuss an algorithm which combines the three ideas: symbolic unification, numeric function evaluation, and inheritance.
4. The Constraint Satisfaction Approach to Unification.

We use the Constrained Unification algorithm introduced in [ROAC 89] as a basis for our unification algorithm. In this chapter we discuss both the algorithms. Before this, we present some definitions that will help in understanding the algorithms better.

Objects.

The universe consists of objects. Objects are 'things' and can be represented by atoms e.g cat, '+'.

Functor objects.

Objects with attributes can be defined as functor objects, e.g. [student sriram cs], [+ 3 [- 6 4]].
Interpreted Functors.

[ROAC 89]
A functor associated with a function on a particular domain is an interpreted functor. For example, $+, -, \times$ are interpreted functors.

Compatible Functors.

[ROAC 89]
Two functors are said to be compatible if

1. both are uninterpreted functors and are syntactically identical or
2. both are interpreted functors with a common domain and the equality and inequality of terms that have these functors as their main functors is decidable.

Terms.

[ROAC 89]
A term may be defined as follows:

1. A variable is a term.
2. An object is a ground term.
3. If $f$ is an $n$-ary uninterpreted functor and $T_1, \ldots, T_n$ are terms, then $[f T_1, \ldots, T_n]$ is a term, called a functor term.
4. If $f$ is an $n$-ary interpreted functor and $T_1, \ldots, T_n$ are terms, that are

- variables

- objects from the domain of interpretation of $f$

- terms of the form $[g X_1, \ldots, X_m]$, where the functors $f$ and $g$ are compatible, and if the terms $X_1, \ldots, X_m$ contain functions, the functors are also compatible with $f$ and $g$.

**Constrained Terms.**

[ROAC 89]

A term as defined by sub-clause 4 of the definition of terms is called a constrained term, and any variable that occurs in a constrained term is also a constrained term.

**Interpreted Predicates.**

[ROAC 89]

An interpreted predicate is an $n$-place predicate symbol associated with a relation on some particular domain. e.g. $=, \neq, \leq$. 

4. The Constraint Satisfaction Approach to Unification.
Constraints.

[ROAC 89]
A constraint is a relation of the form \(< p, t_1, t_2 >\) where \(p\) is an interpreted predicate symbol and \(t_1\) and \(t_2\) are constants from the intended domain or variables ranging over the intended domain or constrained terms whose main functors are the interpreted function symbols of the intended domain. For example, \(< \neq, x, y >\).

Constrained Unification.

Given a set of ordered pairs \(\{(s_1, t_1), \ldots, (s_n, t_n)\}\) and a set of constraints \(S\) where \(s_i\) and \(t_i\) are first order terms and the variables in \(s_1\) through \(s_n\) do not occur in any of the terms \(t_1\) through \(t_n\), decide whether there exists a substitution/solution \(\sigma\) and a consistent set of constraints \(S'\) such that for all \(i\),

- either \(s_i \sigma = t_i \sigma\)
- or \(S' = S \cup \{< \neq, s_i, \sigma, t_i, \sigma >\}\)

Let \(S\) be the set of constraints. The set is partitioned into disjoint subsets \(S_1, \ldots, S_m\) such that

- If the interpreted function symbols occurring in the terms of the two predicates are compatible, then the two predicates belong to the same sub-set and
- two constraints that do not satisfy the above property belong to different sub-sets.

4. The Constraint Satisfaction Approach to Unification.
Thus the set of constraints is partitioned into equivalence classes. The constraints in each of these classes are solved and the values propagated to other constraints until all the constraints are consistent or an inconsistency is found.

**Constrained Unification Algorithm**

Let \( T = \{ (s_1, t_1), \ldots, (s_n, t_n) \} \). Set \( \sigma \) (the set of substitutions) to \( \phi \). The set of constraints \( S \) initially may or may not be empty. The following steps produce the substitution \( \sigma \) and the consistent set of constraints \( S' \), if they exist.

1. If \( T \) is empty, check the consistency of the constraints in \( S \). If they are consistent, output \( \sigma \) and \( S \) and stop, else there is no solution.
2. Apply the \( \sigma \) to all the terms in \( T \) simultaneously.
3. Remove a pair \( (s_i, t_i) \) from \( T \).
4. If \( s_i \) and \( t_i \) are unconstrained terms and either \( s_i \) or \( t_i \) is a variable and the variable term does not occur in the non-variable term then add either \( s_i = t_i \) or \( t_i = s_i \) as the case may be.
5. If \( s_i \) and \( t_i \) are both unconstrained terms of the form \( f(a_1, \ldots, a_m) \) and \( f(b_1, \ldots, b_m) \) then add the pairs \( (a_1, b_1), \ldots, (a_m, b_m) \) to \( T \) and go to step 1.
6. If \( s_i \) and \( t_i \) are both constants, they must be identical or else stop (no solution).
7. If one of \( s_i \) or \( t_i \) is a variable, and the other is a constrained term, add the constraint \( < =, s_i, t_i > \) to \( S \) and check the consistency of the set \( S \). If \( S \) becomes inconsistent then there is no solution. If \( S \) remains consistent and some variables that occur in the constraint are instantiated or a substitution has been found for them, add the substitutions to \( \sigma \) and go to step 1.

4. The Constraint Satisfaction Approach to Unification.
8. If both are constrained terms of the form $f(\ a\text{, }\ldots\text{, }a_p\ )$ and $g(\ b\text{, }\ldots\text{, }b_q\ )$ then $f$ and $g$ must be compatible functors. If not, there is no solution, stop. If yes, then add the constraint $<\ =\ f(\ a\text{, }\ldots\text{, }a_p\ ),\ g(\ b\text{, }\ldots\text{, }b_q\ )\ >$ to $S$. ($p$ is not necessary equal to $q$). If $S$ becomes inconsistent, then there is no solution. If $S$ remains consistent, add the substitutions (if any) found for the variables in $S$ to $\sigma$ and go to step 1.

Constrained Unification is defined on a single universe. We are considering a universe that is divided into several subuniverses, therefore we modify this algorithm to perform in this structured universe.

**Types.**

The universe of objects is divided into sets called types. The ‘subset of’ relation induces a partial ordering on these sets called a hierarchy. The special types **top** and **bottom** denote the set of all objects and the empty set respectively. All types are a ‘subset of’ top, and bottom is a ‘subset of’ all types.

**Typed Terms.**

Terms can be extended to include a type constraint. A typed term may be defined as follows, where $t$ is a type:

1. **variable**: $t$ is a typed term,
2. **object**: $t$ is a typed term, called a **ground typed term**,
3. if $T_1, \ldots, T_n$ are typed terms, then $[f T_1, \ldots, T_n]$: $t$ is a typed term,

4. all other terms are typed terms, equivalent to term: top.

We will refer to typed terms as terms from now on. We redefine interpreted functors to include typed terms.

**Interpreted Functors.**

Functors associated with specific functions on particular domains are called interpreted functors. Interpreted functors return values of a particular type: called the return type of the functor.

We define a function:

$\text{Interpret}(t)$, where $t$ is a term. This returns a term resulting from the evaluation of all interpreted functor terms in the term. For example,

$[+ 5 3]$ will result in the constant 8,

$[- X 3]$, where $X$ is a variable, will remain $[- X 3]$.

**Typed Unification Algorithm.**

We modify the constrained unification algorithm to handle unification of typed terms.

We need to add a few more sets to the algorithm.

Let $R$ be the set of all constrained variables.

4. The Constraint Satisfaction Approach to Unification.
Algorithm.

Let \( T = \{(s_1, t_1), (s_2, t_2), \ldots\} \) be the set of pairs of terms to be unified. Let \( \sigma \) be the set of substitutions. Set \( \sigma \) to \( \emptyset \). Let \( S \) be the set of constraints. Initially \( S \) and \( R \) may or may not be empty.

In case of numerical constraints, if the number of constraints is not equal to the number of variables in the constraints it is not possible to obtain a solution. Thus, if the constraints are consistent we keep them in \( S \), adding constraints to \( S \) until we get a solution. If the number of constraints are not enough to obtain a solution finally, we return the simplified constraints as the solution. Therefore, it is possible for \( S \) and \( R \) to be non-empty at the start of this algorithm.

1. If \( T \) is empty, check the consistency (described later) of the constraints in \( S \). If they are consistent stop, else there is no solution and unification has failed.
2. Apply \( \sigma \) to all terms in \( T \) simultaneously.
3. Remove a pair \((s_i, t_i)\) from \( T \).
4. If \( s_i \) is an atom and
   a. \( t_i \) is an atom, then they must both be identical, else stop because unification has failed.
   b. \( t_i \) is an unconstrained variable. Then if \( s_i \) is an element of the type of \( t_i \) then add \( t_i = s_i \) to \( \sigma \) and goto 1.
   c. \( t_i \) is a constrained variable. Then if \( s_i \) is an element of the type of \( t_i \) then add \( < =, t_i, s_i > \) to the set of constraints \( S \). Check the consistency of \( S \). If \( S \) becomes inconsistent, then there is no solution and unification fails. If \( S \) remains consistent, then add all new substitutions (if any) to \( \sigma \) and goto 1.

4. The Constraint Satisfaction Approach to Unification.
d. \( t_i \) is a functor term, then stop because unification has failed.

5. If \( s_i \) is an unconstrained variable and
   
a. \( t_i \) is an unconstrained variable. Then let \( t' \) be the glb of the types of \( s_i \) and \( t_i \). If \( t' \neq \text{Bottom} \) then set the type of \( t_i \) to \( t' \), add \( s_i = t_i \) to \( \sigma \), else stop because unification has failed.

b. \( t_i \) is an unconstrained functor term. Then let \( t' \) be the glb of the types of \( t_i \) and \( s_i \). If \( t' \neq \text{Bottom} \) then let the type of \( t_i \) be \( t' \). Add \( s_i = t_i \) to \( \sigma \). Else stop because unification has failed.

c. \( t_i \) is a constrained variable. Add the constraint \( < =, t_i, s_i > \) to \( S \). Add \( s_i \) to \( R \). Check the consistency of \( S \). If \( S \) becomes inconsistent then stop, because unification has failed. If \( S \) remains consistent, add all new substitutions (if any) to \( \sigma \).

d. \( t_i \) is a constrained functor term. Then let \( t' \) be the GLB of the types of \( t_i \) and \( s_i \). If \( t' \neq \text{Bottom} \) then add \( < =, s_i, \text{Interpret}(t_i) > \) to \( S \). Add \( s_i \) to \( R \). Check the consistency of \( S \). If \( S \) becomes inconsistent then stop, because unification has failed. If \( S \) remains consistent, add all new substitutions (if any) to \( \sigma \).

6. If \( s_i \) is an unconstrained functor term and
   
a. \( t_i \) is also an unconstrained functor term. Then let \( t' \) be the glb of the types of \( t_i \) and \( s_i \). If \( t' \neq \text{Bottom} \), set the type of \( s_i \) to \( t' \). Add \( t_i = s_i \) to \( \sigma \) and goto 1. Else stop because unification has failed.

b. \( t_i \) is a constrained variable. Then let \( t' \) be the glb of the types of \( t_i \) and \( s_i \). If \( t' \neq \text{Bottom} \), set the type of \( s_i \) to \( t' \). Add \( t_i = s_i \) to \( \sigma \). Add all variables in \( s_i \) to \( R \) and goto 1. Else stop because unification has failed.

c. \( t_i \) is a constrained functor term. Then let \( t' \) be the glb of the return types of \( t_i \) and \( s_i \). If \( t' \) is \( \text{Bottom} \) then stop because unification has failed. Otherwise, add
the constraint \(<\ =\ ,\ s_i,\ \text{Interpret}(t_i)\ >\) to \(S\). Check the consistency of \(S\). If \(S\) becomes inconsistent then stop because unification has failed. If \(S\) remains consistent, add all new substitutions (if any) to \(\sigma\).

7. If \(s_i\) is a constrained variable and
   a. \(t_i\) is a constrained variable. Then let \(t'\) be the glb of the types of \(s_i\) and \(t_i\). If \(t'\) \(\neq\) \textbf{Bottom} then form the constraint \(s_i = t_i\) and add it to \(S\). Check the consistency of \(S\). If \(S\) becomes inconsistent then stop because unification has failed. If \(S\) remains consistent, add all new substitutions (if any) to \(\sigma\). If \(t'\) is \textbf{Bottom}, then stop because unification has failed.
   b. \(t_i\) is a constrained functor term. Then let \(t'\) be the glb of the return type of \(t_i\) and the type of \(s_i\). If \(t'\) is \textbf{Bottom}, then stop because unification has failed. Otherwise add the constraint \(<\ =\ ,\ s_i,\ \text{Interpret}(t_i)\ >\) to \(S\). Check the consistency of \(S\). If \(S\) becomes inconsistent then stop, because unification has failed. If \(S\) remains consistent, add all new substitutions (if any) to \(\sigma\).

8. If \(s_i\) is a constrained functor term and
   a. \(t_i\) is also a constrained functor term. Let \(t'\) be the glb of the return types of \(s_i\) and \(t_i\). If \(t'\) is \textbf{Bottom}, then stop because unification has failed. Otherwise, add the constraint \(<\ =\ ,\ \text{Interpret}(s_i),\ \text{Interpret}(t_i)\ >\) and add it to \(S\). Check the consistency of \(S\). If \(S\) becomes inconsistent then stop, because unification has failed. If \(S\) remains consistent, add all new substitutions (if any) to \(\sigma\).

For the above algorithm, we need to have a method to check the consistency of a set of constraints. To do this, we apply the method described in constrained unification [ROAC 89]. We check for consistency of \(S\) when a new constraint \(<\ =\ ,\ t_i,\ t_2\ >\) is added to \(S\). Consider the set \(M\) of all constraints that \(t_i\) and \(t_2\) occur in. We remove a set of constraints that form an equivalence set, or a single constraint from \(M\). We prune the
domains of each variable in the constraint or constraints we removed, by removing all values for which no values exist in the other variables for the constraint or constraints to be consistent. We use equivalence sets of constraints because efficient algorithms exist for checking the consistency of linear equations and inequalities etc.

**Completeness and Termination Properties:**

**Completeness:** This property is connected with the completeness of the consistency checking algorithm. For linear constraints, the solver will always return a solution, but not for non-linear constraints.

**Termination:** This algorithm terminates when T is empty. T is a finite set of pairs of terms. If the consistency checking algorithm terminates, then this algorithm terminates.
5. Incorporating Arithmetic into Unification.

Logic Programming combines expressive and computational power with a simple and elegant semantics [LASS 87]. Prolog is a logic programming language that achieves reasoning by syntactic inferencing techniques. The semantics of Prolog are defined within the context of the Herbrand Universe, the set of all first order terms that can be formed from the constants and function symbols in the Prolog program. Unification, used in resolution, is basically a form of syntactic matching. There is no function evaluation. In unification over the Herbrand universe, only those terms that are syntactically equivalent can be unified together but not two terms that are semantically identical yet syntactically distinct. Computations on symbolic domains can be efficiently accomplished by syntactic matching but not numeric domains. Numeric domains have algebraic operations associated with them. Computations on numeric domains would be more efficient if they were achieved by evaluating these operations rather than by syntactic inferencing.

To incorporate arithmetic, it is necessary to treat functions as objects of computation. During unification, it is necessary to compute functions and not just syntactically match the two terms. Thus it is necessary to generalize unification to take care of quantitative reasoning as well as syntactic inferencing. Constraint satisfaction is a model for quantitative and qualitative reasoning. Therefore, we use it as the model of computation.
The time complexities for solving simultaneous arithmetic equations were used to
determine the kinds of functions incorporated into unification. These complexities are
discussed in [DAVI 87]. It was necessary to consider functions that can be efficiently
solved. The complexity of handling non-linear problems is known to be NP-Hard,
while equations with transcendental functions are, in general, undecidable. Special
cases of non-linear systems could have been handled easily like systems that be-
come linear due to variables becoming instantiated. Solving the system of equations
could be delayed until it became linear. In this case, the programmer is responsible
for ensuring that the system does become linear otherwise the solving would be
suspended indefinitely. Thus, we restrict the systems we consider to linear systems
with polynomial solution techniques.

Implementation Details.

In Prolog each goal is solved one at a time. If the goal is satisfied the next goal is
tried, otherwise Prolog backtracks to the previous backtrack point and follows an-
other branch of the solution tree. This is the case even if the goal is numerical.
Quantitative variables often have an infinite range of values and this process could
be very time consuming. By trying to satisfy all the numerical goals in the clause si-
multaneously, it is possible to find an answer in much less time. Thus, incorporating
arithmetic involves solving systems of numerical constraints. These constraints are
basically equations consisting of different functions applied to variables. Thus these
systems can be considered as systems of equations. Only linear equations are han-
dled because of the complexity of solving non-linear equations.

5. Incorporating Arithmetic into Unification.
Linear inequalities can be handled by linear programming techniques like the simplex method [DANT 51]. First all inequalities have to be converted to equations by adding slack variables to $\leq$ constraints and excess variables to $\geq$ constraints. These variables are constrained to be $\geq 0$. Handling $\neq$ type constraints involves considering the constraint as a $<$ type first and then a $>$ type. The slack and excess variables in this case have to be strictly $> 0$. The dual simplex method was considered to implement this. Linear inequalities, however, were not implemented.

The system of equations is built up while the Prolog runtime machine is trying to satisfy the numerical goals in the body of the clause. Equations are created when a numerical goal has to be satisfied, i.e., when the equality predicate is used with numeric arguments. Also, when two terms containing variables are unified, and one of the variables is a constrained variable, an equation of the form

\[
\text{variable} = \text{constrained variable}
\]

is created. Whenever a variable is instantiated after the system has been set up, the equation

\[
\text{variable} = \text{binding}
\]

is added to the set of constraint equations. The system is solved before satisfying the next goal with a user-defined predicate.

An expression simplifier is applied to all constraints before storing them. The equations are stored in a canonical form, as the sum of products. The simplifier handles only linear expressions and will generate an error message on encountering a non-linear expression. For convenience, the expressions with a binary minus operator are rewritten with a unary minus. For example,

\[
(-5 \times) \text{ is simplified to } (+ -5 \times).
\]

5. Incorporating Arithmetic into Unification.
The expression simplifier handles addition, multiplication, subtraction and division operators. It uses a recursive algorithm on each operand of each operator. For example,

\[ (* 3 (+ x (* 2 y))) \text{ is simplified to} \]
\[ (+ (* 3 x) (* 6 y)). \]

Division by variables is not handled since it complicates the check for linearity of the constraint.

Linear algebra techniques are used to solve the system of linear equations. It is not always possible to obtain a unique solution for the variables in the equations. Sometimes, only the consistency of the system is checked. When the number of variables is less than or equal to the number of equations, the solution for the variables is found, if the system is consistent and of full rank. When the number of variables is greater than the number of equations or the system is rank deficient, the system may have infinitely many solutions if the system is consistent. In this case, we will only conclude that the system is consistent or inconsistent. If at the end of a query, unique solutions for the variables do not exist, the constraints on the variables are output as part of the answer. For example, consider

\[ (+ (+ a (* 2 b)) (+ c -6)) = 0 \]
\[ (+ (+ b c) -3) = 0 \]

As this is a system of two equations in three unknowns, it is not possible to obtain a unique solution. By inspection, we know that a solution \(a = 1, b = 2, c = 1\) exists. The system is consistent but under-determined, hence the answer returned for the system is the same two constraints.
The case when the number of equations is greater than the number of variables, but the equations are not all independent and the system is rank deficient, will be treated in the same way as the undetermined case as both are rank deficient.

The method used to solve the systems of equations is based on the QR decomposition method. In this method, the coefficient matrix is decomposed into the product of an orthogonal matrix and an upper triangular matrix. This decomposition reveals any rank deficiency in the coefficient matrix. If a unique solution exists, it is possible to obtain it by backsolving.

Suppose the system to be solved is \(Ax = b\), where

- \(A\) is the \(n \times p\) matrix of coefficients in the equations,
- \(n\) is the number of equations,
- \(p\) is the number of variables,
- \(x\) is the \(p \times 1\) vector of variables,
- \(b\) is the \(n \times 1\) vector of right-hand sides of the equations.

The QR method is used to decompose \(A\) such that

\[ A = QR, \]

where

- \(Q\) is an orthogonal \(n \times n\) matrix,
- \(R\) is an upper triangular \(n \times p\) matrix.

The equation now becomes

\[ QRx = b. \]

Thus, the solution \(x\) is obtained from the equation

\[ Rx = Qr b. \]
This method was preferred over other linear algebra techniques, even standard Gaussian elimination, because though it has the same complexity, it is more accurate and can also be used incrementally. This means that once a system is tested for consistency, the solver will work with the constraints in a solved form along with the added constraints. In Prolog the system of constraints is continually updated and hence this is more efficient.

The equations are created by unifying two terms. The expression simplifier manipulates the equation and puts it into a sum of products form. All the constraints or equations that are currently to be solved are stored on a constraint stack in this form (see Figure 1). Each constraint is stored as the sum of coefficient-variable pairs. LINPACK [DONG 80] subroutines are used to perform the mathematical computations. As they are available in FORTRAN, and since FORTRAN manipulates matrices efficiently, the system is converted to a matrix of coefficients when it has to be solved. The columns of the matrix correspond to the variables in the equations. Each variable is associated with a column number and whenever a new variable is introduced into the system, a new column number is generated. This matrix is not discarded until the query is answered.

One of the advantages of the QR method is that equations can be added to the system and the decomposition can be incrementally updated. This increases the speed of solving the system after the addition of equations because the entire decomposition need not be redone. Incremental update is used when the system is underdetermined (although it is applicable to the overdetermined case also). This is because in case of a overdetermined system, we find a unique solution when we solve the system. The variables are instantiated and therefore, additional constraints in the

5. Incorporating Arithmetic into Unification.
5. Incorporating Arithmetic into Unification.
same variables can be evaluated. The system is solved every time a goal with a user-defined predicate has to be solved. As the system is under-determined, solving the system results in a check for consistency and no solutions are obtained. After the system is solved once, the matrix $R$ is maintained in upper triangular form. The new equation is added to the bottom of $R$ as it is. Then, by using Givens rotations [DONG 80], $R$ is modified so that it remains upper triangular. The same rotations are applied to $Q'$ to modify it to $Q''$.

The system to be solved now is

$$R'x = Q'b.$$

Since Prolog's search technique uses symbolic inferencing, it may be necessary to backtrack to a point to try to find a solution whenever the current set of goals cannot be satisfied. This is the most time consuming operation in Prolog, and hence it has to be handled efficiently. Therefore the system solver needs to backtrack efficiently. A backtrack point is created when it is possible to have more than one solution. The number of equations at that point is stored in the backtrack point. On backtracking, the coefficient matrix corresponding to the current system is discarded and the matrix corresponding to the remaining number of equations is rebuilt. The number of remaining equations is obtained from the information stored at the backtrack point. Instead of rebuilding the equation matrix, we could have restored the matrix by incrementally deleting the equations. Incremental deletion of equations, however, was found to be involved and inefficient. This was because rows corresponding to the equations have to be deleted and the effect of transforming $R$ to $R'$ needs to be undone. Information to undo these transformations must also be stored. If the deletion of a row causes a variable in the system to disappear, the column corre-
sponding to the variable also has to be deleted. An alternate method to rebuilding the matrix is to store the current coefficient matrix at each backtrack point. This is space expensive and can be considered only when there is a large amount of memory available. The coefficient matrix is basically an upper triangular matrix and could be stored using sparse matrix techniques. For reasons of simplicity, these techniques have not been used.

The LINPACK library of subroutines was used to perform the mathematical manipulations of the coefficient matrix, i.e., the decomposition of the coefficient matrix A, the back substitution to obtain the solution, the application of Givens rotations to the upper triangular matrix R, and the corresponding modifications to Q'.

Using this method, we are guaranteed to obtain a solution (if it exists) for systems that have the number of equations greater than or equal to the number of variables and are of full rank. In case of underdetermined systems or rank deficient systems we return the unsolved constraints as the answer, if the system is consistent. Hence, this method is complete. If we were handling non-linear constraints, then this would not be so, because of the possibility that equations might never become linear, in which case the solution of the equations would be delayed indefinitely.
6. Incorporating Type Inheritance into Unification.

In Prolog, terms are defined over a single universe, i.e., the Herbrand Universe. A variable can represent any object in this universe. In unification over a single universe, the search space consists of the entire universe.

In general, the domain under consideration is often structured and can be divided into several subuniverses. These subuniverses may completely overlap, may partially overlap, or may be disjoint.

Types are sets of objects. In fact, a type denotes a subuniverse and is the set of objects in that subuniverse. As subuniverses may be contained in one another, the ‘subset of’ relation induces a partial ordering on types. A type \( t_i \) that is a ‘subset of’ \( t_j \) inherits all the properties of \( t_j \).

Prolog uses resolution to solve goals. Unification is the heart of resolution. During unification, ‘taxonomic’ information can be used to compute unifications more efficiently. Unification with inheritance of types can reduce the number of resolution steps required to solve a goal. By considering a structured universe, it is also possible to prune the search space since the result is constrained to belong to a particular type.
Types correspond to subuniverses from the domain of discourse. A term in Prolog represents the set of objects from the universe that can unify with the term. By adding types, terms are constrained to belong to a particular type. Thus, terms now represent the set of objects that unify with the term and are members of the type of the term. Unification in resolution is the process of making two terms syntactically equal to each other by simultaneously substituting terms for all variables in the two terms. Unification of typed terms consists of forming the intersection of the sets of objects in the universe represented by the two terms. This means that it is a subset of the set that represents the intersection of the sets corresponding to the types of the two terms. To keep unification as general as possible, we choose the largest set that represents the intersection of the two types. This is called the greatest lower bound (glb) of the two types. Thus the most general unifier is the syntactically unified term that belongs to the type of the glb.

Since unification is the key component of resolution, computations in Prolog require several unifications. Hence it is necessary for unification with types to be performed efficiently.

**Implementation Details.**

Objects in the universe can be atoms or functor objects. For example, pencil, [dog apso miniature]. Terms represent sets of objects. The functor term [dog ?name miniature] represents the set of all functor objects that are miniature dogs. A functor object makes it possible to define objects with attributes. The functor object [dog

6. Incorporating Type Inheritance into Unification. 44
?name ?type] represents the complete set of dogs. If we need to define a particular property for miniature dogs, we could declare the following rule

(( hyper [dog ?name miniature] )).

In this implementation, it is necessary to specify terms as elements of a particular type, therefore declarations must be specified before any queries are processed. The user can make the following type declarations:

1. A type is a subtype of a type.

2. A term is a member of a type.

A type \( t_i \) is said to be reachable from \( t_j \) if \( t_i \) is a subtype of \( t_j \) or if \( t_i \) is a subtype of some \( t_k \), and \( t_k \) is reachable from \( t_j \). From declarations of type 2, an object is said to be a direct element of the type. If the type \( t_i \) is reachable from \( t_j \) by more than one link, then the objects of \( t_i \) are said to be indirect elements of \( t_j \).

The type structure explained above is a directed graph consisting of types as nodes. There exists an arc from \( t_i \) to \( t_j \) if \( t_j \) is a subtype of \( t_i \). The graph must not have any cycles or loops, and user input is validated to avoid this.

The type structure is stored in a table (see Figure 2) indexed on type numbers. For each type there is a list of atoms or objects and a list of functor terms which are the direct elements of that type, and a list of subtypes. The list of atoms is a linked list sorted on hash code numbers. The list of functor terms is also sorted on the hash code of the functor symbol. The list of subtypes is stored as a bitset, which means

6. Incorporating Type Inheritance into Unification.
Figure 2. Data Structures for Adding Types with Inheritance to Unification.
there is a bitset associated with each type in the system. For the bitset associated with a type, the ith bit is set if the ith type is a subtype of this type.

The main operation during unification is the 'intersection of types' operation to find the glb. The glb is the type that represents the complete intersection of the two types to be unified.

To facilitate the process of forming the glb, we wish to have all intersecting elements of the two types as elements of a common subtype. This is called the condition of intersection.

To be able to find all common subtypes, it is necessary to take a closure of the declared structure. This means that for all types t, all types that are reachable from t are added to the subtypes of t.

Since subtypes of a type are stored in bitsets corresponding to each type, taking the logical 'AND' (the intersection) of the bitsets computes the set of all common subtypes of the two types. Creation of a new type that has all these common subtypes as its subtypes corresponds to the process of finding the glb. This is because the new type represents the complete intersection of the two types (see Figure 2). The glb's are also referred to as intersection types.

Since it would be extra work to create the glbs of all possible pairs of types, some of which would never be used, we create glb's as and when required. Also since it is possible that the same two types are involved in unification more than once, we remember the result of computing the glb by storing the type numbers of the glb's. This could have been achieved using a two-dimensional matrix indexed on type
numbers, but the matrix is symmetrical and we are only interested in the lower triangular part. Therefore, we have used a single dimensional array to store the glb’s that have been computed. The lower triangle of the matrix is used because it is easier to extend the matrix when new intersection types are created. It is possible to create glb’s incrementally because the process of forming a glb is fast once the condition of intersection holds. This is because the process consists of taking the intersection of the bitsets of subtypes of the two terms and adding a new type to hold the intersection.

All data structures connected with the implementation of types are dynamically created if and when the first type declaration is encountered. Programmers must declare types before any query using the type structures. There is no limit on the number of types that can be handled, because all the structures can be extended as the type structure expands.

The Prolog machine has a stack called the trail on which backtracking information is stored. All unifications that result in the type of a term being changed are trailed, i.e., the address of the term and its previous type value is stored. If a series of unifications are applied to the same term, each change is trailed so that it is possible to backtrack to any place in the run.

This implementation fits in well with the WAM [WARR 83] concept because we have used the same basic data structures and representation of terms. The incorporation of types and inheritance is basically an extension of the WAM concept that is the design for a standard Prolog compiler. In Appendix C we describe the modifications.
made to the WAM and also the new instructions to be added to the code generated by the WAM.

The next chapter discusses the theory of adding types to unification and also the correctness of this implementation.
7. Theory and Correctness of the Types Implementation.

Definitions.

In this chapter we discuss the theory behind the types concept and show the correctness of our implementation. For the convenience of the reader we will repeat some definitions.

Objects.

The universe consists of objects. Objects are ‘things’ and can be represented by atoms, e.g. cat, and functors objects, e.g. [student sriram cs].

Terms.

A term may be defined as follows:

1. A variable is a term.
2. An object is a ground term.

3. If f is an n-ary functor and \( T_1, \ldots, T_n \) are terms, then \([f \, T_1, \ldots, T_n]\) is a term called a functor term.

**Variable Substitutions.**

A variable substitution is a finite set of the form \( \{ v_1 / T_1, \ldots, v_n / T_n \} \), where each \( v_i \) is a distinct variable, and each \( T_i \) is a term. In a variable substitution, each \( v_i / T_i \) is called a binding for the variable \( v_i \).

Let \( T \) be a term and let \( V = \{ v_1 / T_1, \ldots, v_n / T_n \} \) be a variable substitution. Then \( V(T) \), the result of applying the variable substitution \( V \) to the term \( T \) is the term obtained by simultaneously replacing each occurrence of the variable \( v_i \) by the term \( T_i \). If all the \( T_i \) are ground terms, then \( V \) is called a ground substitution.

**Mappings.**

We consider a mapping \( P : S \rightarrow T \), to be a set of ordered pairs \( \{ (s_1, t_1), (s_2, t_2), \ldots \} \), where for all \( i, s_i \in S \) and \( t_i \in T \). Therefore, we can speak of taking a subset of a mapping and adding elements to a mapping. \( P \) is a function mapping if all the \( s_i \) are distinct, in which case we define

\[ P(s_i) = t_i \text{ to mean } (s_i, t_i) \in P. \]

7. Theory and Correctness of the Types Implementation.
The Mapping M.

Terms represent sets of objects. The objects of this set are the ground terms obtained by applying all possible ground substitutions to the term.

Consider a function mapping $M: \text{term} \rightarrow \text{set of objects}$, then

1. $M(\text{variable}) = H$, where $H$ denotes the set of all objects in the universe.
2. $M(\text{object}) = \{\text{object}\}$
3. $M(\text{F}) = \{V_1(F), V_2(F), \ldots\}$, where $V_1, V_2, \ldots$ are all possible ground substitutions for the variables in the functor term $F$.

We will occasionally use the form $M(R)$, where $R$ is a set of terms. In this context, $M(R) = \bigcup_{T \in R} M(T)$.

Unification.

Unification is a function mapping from a set of terms to a single term, called the unifier of the terms in that set. The mapping is partial since a unifier does not always exist.

Given $M$, unification can be defined as mapping a set of terms $\{T_1, \ldots, T_n\}$ to a term $T'$ such that $M(T') = M(T_1) \cap \ldots \cap M(T_n)$.
Types.

A type denotes a set of objects. The members of a type must be explicitly declared. Each type has a name, which is an atom.

Simple Type Graph.

A simple type graph, \( G = (N, \leq, T, \bot) \), is an ordered 4-tuple, where \( N \) is a finite set of types, \( \leq \) is a binary relation on \( N \), and \( T \) (top) and \( \bot \) (bottom) are special types for which the following hold:

1. \( T \in N \), \( \bot \in N \);
2. for all \( t \in N \) : \( \bot \leq t \);
3. for all \( t \in N \) : \( t \leq T \);
4. for all \( t \in N \) : \( t \leq t \).

Closed Type Graph.

A closed type graph, \( G = (N, \leq, T, \bot) \), is a simple type graph in which:

1. for all \( t_1, t_2, t_3 \in N \) : (\( t_1 \leq t_2 \) and \( t_2 \leq t_3 \)) implies \( t_1 \leq t_3 \), i.e., transitivity holds.
2. for all \( t_1, t_2 \in N \) : (\( t_1 \leq t_2 \) and \( t_2 \leq t_1 \)) implies \( t_1 = t_2 \), i.e., \( t_1 \) and \( t_2 \) are the same type and no cycles exist in the graph.

7. Theory and Correctness of the Types Implementation.
**Type Structure.**

A type structure $S = (N, B, \leq, T, \bot, \theta)$ is an ordered 6-tuple, where $(N, \leq, T, \bot)$ forms a closed type graph, $B$ is a set of objects, and $\theta$ is a mapping from $N \rightarrow 2^B$. An object $o$ is said to be a member of the type $t$ if and only if $o \in \theta(t)$.

**Closed Type Structure.**

A closed type structure $S = (N, B, \leq, T, \bot, \theta)$ is a type structure for which, if $o$ is an object and $t_1$ and $t_2$ are types, then

$$(o \in \theta(t_1) \text{ and } t_1 \leq t_2) \implies o \in \theta(t_2).$$

**Complete Type Structure.**

A complete type structure $S = (N, B, \leq, T, \bot, \theta)$ is a closed type structure in which:

For all $t_1, t_2 \in N$, there exists $t' \in N$:

$$\theta(t') = \theta(t_1) \cap \theta(t_2)$$

$t'$ is called the intersection type for $t_1$ and $t_2$.

Given a closed type structure $S = (N, B, \leq, T, \bot, \theta)$, we construct a complete type structure $S' = (N', B, \leq', T', \bot', \theta')$. Let the elements of $N$ be $t_1, \ldots, t_n$.

1. $N'$ consists of all $t_s$ where $s$ ranges over elements of $\{0,1,\ldots,2^{2n}\}$.
2. $\leq'$ is defined as

$$t_s \leq' t_u \text{ iff } u \leq s$$

7. Theory and Correctness of the Types Implementation.
3. \( T' = t_s \)

4. \( \bot' = t_{(1..n)} \)

5. \( \theta' \) is defined as

\[
\theta'(t_t) = B, \text{ if } s = \phi \\
\text{ else } \cap_{i \in s} \theta(t_t)
\]

To show that \( S' \) is a complete type structure,

let \( G' = (N', \leq', T', \bot') \). Then

1. \( G' \) is a simple type graph, because

\[
T' = t_s \in N' \\
\bot' = t_{(1..n)} \in N'
\]

For all \( t_t \in N' \): \( \bot' = t_{(1..n)} \leq' t_t \) because \( s \subseteq \{1..n\} \)

For all \( t_t \in N' \): \( t_t \leq' T' = t_s \) because \( \phi \subseteq s \)

For all \( t_t \in N' \): \( t_t \leq' t_t \) because \( s \subseteq s \)

2. \( G' \) is a closed type graph because

For all \( t_t, t_u, t_v \in N' \): \( (t_t \leq' t_u) \) and \( (t_u \leq' t_v) \)

\[ = (u \subseteq s) \text{ and } (v \subseteq u) \]

which implies \( v \subseteq s \)

\[ = t_t \leq' t_v \]

For all \( t_t, t_u \in N' \):

\[ (t_t \leq' t_u) \text{ and } (t_u \leq' t_t) \]

\[ = (u \subseteq s) \text{ and } (s \subseteq u) \]

implies \( u = s \), and therefore \( t_u = t_t \)

3. \( S' \) is a closed type structure because

For some object \( o, o \in \theta'(t_t) \) and \( t_t \leq' t_u \)

7. Theory and Correctness of the Types Implementation.
= \sigma \in \cap_{i \in \mathbb{N}} \theta'(t_i) \text{ and } u \subseteq s, \\
\text{but, } \cap_{i \in \mathbb{N}} \theta'(t_i) \subseteq \cap_{i \in \mathbb{N}} \theta'(t_i) \\
\text{thus } \sigma \in \cap_{i \in \mathbb{N}} \theta'(t_i) \\
= \sigma \in \theta'(t_s)

4. For any two types \( t_s \) and \( t_n \), we prove that \( t_{(s\cup r)} \) is their intersection type: 
   
   i.e. \( \theta'(t_{(s\cup r)}) = \theta'(t_s) \cap \theta'(t_n) \) 
   
   \( \cap_{i \in \mathbb{N}} \theta'(t_i) = (\cap_{i \in \mathbb{N}} \theta(t_i)) \cap (\cap_{i \in \mathbb{N}} \theta(t_i)) \) 

   we know that 
   
   \( (s \cup r) = (s - r) \cup (s \cap r) \cup (r - s) \) 
   
   \( s = (s - r) \cup (s \cap r) \) 
   
   \( r = (r - s) \cup (s \cap r) \) 

   but since \( (s \cap r) \cap (s \cap r) = (s \cap r) \), the equality holds.

   Thus \( S' \) is a complete type structure.

For a closed type structure \( S = (N, B, \leq, T, \perp, \theta) \), if \( N \) contains \( n \) types, the corresponding complete structure may contain as many as \( 2^n \) types.

**Typed Terms.**

Terms can be extended to include a type constraint. A typed term may be defined as follows, where \( t \) is a type:

1. (variable: t) is a typed term 
2. (object: t) is a typed term, called a **ground typed term** 
3. if \( T_1, ..., T_n \) are typed terms, then \( ([f \ T_1, ... T_n]; t) \) is a typed term

**7. Theory and Correctness of the Types Implementation.**
4. All other terms are typed terms, equivalent to (term: T)

We will refer to typed terms as terms from now on.

The Mapping M extended to Typed Terms.

If T is a term and t is a type, we define

\[ M(T:t) = M(T) \cap \theta(t) \]

To understand this we extend the concept of a variable substitution to typed variables. A typed variable substitution is a finite set of the form \( V = \{ v_i : t_i / T_i , \ldots , v_n : t_n / T_n \} \), where each \( v_i \) is a distinct variable, each \( t_i \) is a type and each \( T_i \) is a term. A typed variable substitution is said to be 'permissible' if \( \theta(T_i) \subseteq \theta(t_i) \). If all the \( T_i \) are ground terms, then \( V \) is called a ground substitution.

This implies that if \( t \) is a type,

1. \( M(\text{variable}:t) = \theta(t) \)
2. \( M(\text{object}:t) = \{ \text{object} \} \) if object \( \in \theta(t) \),
   otherwise \( \phi \)
3. \( M(F) = \{ V_1(F), V_2(F), \ldots \} \), where \( V_1, V_2, \ldots \) are all permissible ground typed variable substitutions containing bindings for variables in the functor term \( F \).
Partially Typed Terms (PTT).

1. A variable is a PTT.
2. An object is a PTT.
3. If f is an n-ary functor and $T_1, ..., T_n$ are typed terms, then $[f \ T_1 \ ... \ T_n]$ is a PTT.

Unification of Partially Typed Terms.

Just like normal unification, except if both terms are functor terms, in which case, if the functors don’t match unification fails, else unification succeeds iff all pairs of arguments unify using the unification of typed terms given below.

Unification of typed terms.

Let $T_i : t_i$ and $T_z : t_z$ be two terms. If there exists $T' : t'$ such that

$$M(T' : t') = M(T_i : t_i) \cap M(T_z : t_z)$$  \hspace{1cm} (1)

then $T' : t'$ is the unifier of $T_i : t_i$ and $T_z : t_z$.

By the extension of the mapping $M$ to typed terms, we have

$$M(T : t) = M(T) \cap \theta(t)$$

Therefore, from (1) we get

7. Theory and Correctness of the Types Implementation.
\[ M(T') \cap \theta(t') = (M(T_1) \cap \theta(t_1)) \cap (M(T_2) \cap \theta(t_2)) \]

\[ = (M(T_1) \cap M(T_2)) \cap (\theta(t_1) \cap \theta(t_2)) \]

where \( T_1 \) and \( T_2 \) are partially typed terms.

By the definition of unification we get,

\[ M(T_1) \cap M(T_2) = M(T') \]

where \( T_1 \) and \( T_2 \) are partially typed terms.

Since we are dealing with a complete type structure, for all pairs \( t_1 \) and \( t_2 \) there exists an intersection type, \( t \) such that:

\[ \theta(t) = \theta(t_1) \cap \theta(t_2) \]

Thus the intersection type of \( t_1 \) and \( t_2 \) can be used as \( t' \).

**Correctness of the implementation.**

**Declared Type Structure.**

A declared type structure, \( L = (N,R,D,E) \) is an ordered 4-tuple, where

1. \( N \) is a finite set of types that contains the special types top and bottom.
2. \( R \) is a finite set of terms.
3. \( D \) is a mapping from \( N \rightarrow 2^N \). \( D \) maps a type to the set of its subtypes. We introduce the symbol ‘\( \leftarrow \)’ and let \( t_2 \leftarrow t_1 \) mean \( t_2 \in D(t_1) \). We say that \( t_2 \) is directly
reachable from \( t \) if \( t_i \leftarrow t \), and that \( t_i \) is indirectly reachable from \( t \) if \( t_i \leftarrow t \), and \( t_i \) is indirectly reachable from \( t_i \). \( D \) must include

- \( t_i \leftarrow \text{top}, \text{bottom} \leftarrow t_i, t_i \leftarrow t_i \), for all \( t_i \in N \)
- For all \( t_i \), if \( t_j \in N \), \( t_i \) is indirectly reachable from \( t_j \) and \( t_j \) is indirectly reachable from \( t_i \) then \( t_i = t_j \).

4. \( E \) is a mapping from \( N \rightarrow 2^\infty \). \( E \) maps a type to a set of terms representing the objects belonging to that type. We introduce the symbol \( '\infty' \) and let \( T \infty t \) mean \( T \in E(t) \).

**Constructing a closed type graph.**

Given a declared type structure \( L = (N, R, D, E) \) we construct \( L' = (N, R, D', E) \) where,

\[
D' : N \rightarrow 2^\infty \text{ is defined by,}
\]

\[
t_i \in D'(t) \text{ iff } t_i \in D(t) \text{ or } (t_i \in D(t_j) \text{ and } t_j \in D'(t))
\]

A recursive algorithm to construct this closure follows:

Algorithm **CLOSE**(t);

for all \( t_i \in D(t) \) do

  If NOT Visited(t) then

    **CLOSE**(t);

  end;

\[
D'(t) = D(t) \cup D(t_i)
\]

end **CLOSE**.

A call to **CLOSE**(top) will construct the full closure.

7. Theory and Correctness of the Types Implementation.
It is trivial to show that \( G = (N, \leftarrow, \text{top}, \text{bottom}) \) is a simple type graph.

Now \( G = (N, \leftarrow', \text{top}, \text{bottom}) \) is a closed type graph, where

\[
t_i \leftarrow' t_j \text{ means } t_i \in D'(t_j) \text{ because}
\]

1. For all \( t_i, t_2, t_3 \),
   
   if \( t_i \leftarrow' t_2 \) and
   
   \( t_2 \leftarrow' t_3 \), then
   
   \( t_i \in D'(t_2) \) and \( t_2 \in D'(t_3) \). By the construction of \( D' \), \( t_i \in D'(t_3) \)

   Therefore, \( t_i \leftarrow' t_3 \), i.e. transitivity holds.

**Constructing a closed type structure.**

Given a declared type structure \( L = (N, R, D, E) \), where \( G = (N, \leftarrow, \text{top}, \text{bottom}) \) is a closed type graph, we construct a closed type structure, \( S = (N, B, \leftarrow, \text{top, bottom}, \phi) \), where

We extend \( E \) to \( E' \) such that if \( t_i \leftarrow t_j \) then \( E'(t_i) \subseteq E'(t_j) \).

We define \( \in' \) such that, \( T \in' t_i \)

\[
\text{iff } T \in t_i \text{ or } (T \in t_i \text{ and } (t_i \leftarrow t_j))
\]

Let \( B = M(R) \), the set of all possible objects in the Herbrand Universe.

We define \( \phi : N \rightarrow 2^B \), such that:

\[
\phi(t) = M(E'(t)).
\]

\( S \) is a closed type structure because:

1. \( (N, \leftarrow, \text{top, bottom}) \) is a closed type graph.

7. Theory and Correctness of the Types Implementation.
2. \( \phi \) is a mapping from \( N \to 2^N \)

3. \( o \in \phi(t_i) \) means \( o \in M(E'(t_i)) \)

   but since \( t_i \leftarrow t_j \) then \( E'(t_i) \subseteq E'(t_j) \) and therefore \( M(E'(t_i)) \subseteq M(E'(t_j)) \), this implies,

   \[ o \in M(E'(t_j)) \]

   that is \( o \in \phi(t_j) \)

**Constructing a complete type structure.**

Given a declared type structure \( L = (N,R,D,E) \) where \( S = (N,B,\leftarrow, \text{top, bottom, } \theta) \), is a closed type structure, we need to construct a complete type structure \( S' = (N \cup I, B, \leftarrow', \text{top, bottom, } \phi') \). The complete type structure can be constructed by creating a new type \( t \) for each \( t_i, t_j \in N \), such that

\[ \phi(t) = \phi(t_i) \cap \phi(t_j) \]

\( t \) is called the intersection type for \( t_i \) and \( t_j \). One way to find the complete intersection of these two types is to construct the intersection by unifying each term \( T \) such that \( (T, \propto t_i \text{ or } T, \propto^* t_i) \) with each term \( T \), such that \( (T, \propto t_i \text{ or } T, \propto^* t_i) \).

We discovered a more efficient method, performed in two stages. In the first stage, intersections of terms directly attached to the types are created and in the second stage an intersection type is created that holds the total intersection of the two types. By using this method, types \( t' \) such that \( t' \leftarrow t_i \) and \( t' \leftarrow t_j \) are not explored while taking the intersection of \( t_i \) and \( t_j \). After the intersection type \( t \) is created, all such \( t' \) are made reachable from \( t \). In the first stage, a new set of types \( Z \), called **extra types**, is created. We extend the mapping \( D \) to \( D' \), which includes pairs containing members of \( Z \). In the second stage, another set of types \( I \) is created, which represent the true

7. Theory and Correctness of the Types Implementation.
intersection types of N. $D^1$ is also extended to $D^2$ which includes pairs containing members of $I$.

**Stage 1.**

The construction of $Z$ and $D^1$ is accomplished by creating the sets $Z_1,...,Z_n$, where $n = |N|$ and $Z = Z_1 \cup ... \cup Z_n$. (note: If $Z_i = \{\}$, where $i < n$, then each of $Z_{i+1},...,Z_n$ will be empty). To create the set $Z_i$, we need sets $N$ and $Z_{i-1}$.

Let $Z_p$ be the previously created set and $Z_e$ be the set we are currently creating.

Each $t \in Z_p$ represents an intersection of certain types from $N$. The types in $N$ are ordered on type numbers. Let $Max(t)$ be the maximum type in this intersection. $Max(t)$ is used to ensure that duplicate types representing the same intersection are not created.

Algorithm **COMPLETE1**;

Let $Z_p = N$, $Z = \{\}$;

For each $t \in Z_p$ do

- $Max(t) = i$;

While $Z_p \neq \phi$

- Let $Z_e = \{\}$

For every $t \in Z_p$ do

- For $i = Max(t) + 1$ to $n$ do

  - if not($t \leftarrow t$) and not($t \leftarrow t_i$) then

    - create a new type $t_k$

    - for each $T_i \equiv t_i$ and $T_j \equiv t$ such that $T_i$ and $T_j$ unify

      - let unify($T_i, T_j$) $\equiv t_k$

      - add $t_k$ to $Z_e$

      - add $t_k \leftarrow t_i$ and $t_k \leftarrow t$ to $D$

    - let $Max(t_k) = i$.

Let $Z = Z \cup Z_e$

Let $Z_p = Z_e$

CLOSE(top);

end **COMPLETE1**.

7. Theory and Correctness of the Types Implementation.
Let $D'$ denote the mapping $D$ after the algorithm terminates, and $t_i \leftarrow t_x$ mean $t_x \in D'(t_i)$.

At the end of stage 1, we wish to show that the condition of intersection is true. i.e.:

For any set of types $S \subseteq N$, if an object $o \in \cap_{t \in S} M(E(t))$, then there exists a type $t'$, and a term $T$, such that, for all $t \in S$, $t' \leftarrow t$, $T \in t'$, and $o \in M(T)$.

For all $S \subseteq N$: (for all $o \in B$:(there exists $t' \in N$: (for all $t \in S$: $o \in \phi(t)$) implies $o \in \phi(t')$))

Proof. To show that applying algorithm COMPLETE1 yields a structure for which the condition of intersection holds.

Case $|S| = 1$: This is trivial since the single member of $S$ serves as $t'$

Assume the condition holds for all sets of size $< m$.

Case $|S| = m$: Let $S = \{t_1, ..., t_m, t_0\}$.

Then, for $S' = \{t_1, ..., t_m\}$ the condition holds. Consider an object $o \in \cap_{t \in S} M(E(t))$. Let $t'$ be the type such that $o \in M(t')$ and $t' \leftarrow t_x$, for all $t_x \in S'$. If $t_x \leftarrow t'$, then $t_x$ serves as $t'$ if $t' \leftarrow t_x$ then $t'$ serves as $t'$. Otherwise, during the $m$th iteration of algorithm COMPLETE1, an appropriate $t'$ is created such that $t' \leftarrow t'$, and $t' \leftarrow t_x$ and $o \in M(E(t'))$.

Stage 2.

The construction of $I$ and $D_Z$ is accomplished by creating the sets $I, I_1, ..., I_n$, where $n = |N|$ and $I = I_1 \cup \ldots \cup I_n$ (note: if $I_i = \{\}$, where $i < n$, then each of $I_{i+1}, ..., I_n$ will be empty).

To create the set $I_n$, we need sets $N$ and $I_{n-1}$.

Let $I_n$ be the previously created set, and $I_{n-1}$ be the set we are currently creating.

7. Theory and Correctness of the Types Implementation.
Each $t_i \in I$ represents an intersection of some types from $N$. The types in $N$ are ordered on type numbers. Let $\text{Max}(t)$ be the maximum type number (from $N$) in this intersection.

Parents$(t)$ is the set of types $\{t_i\} \in N$, such that
\[
\phi(t) = \cap \phi(t_i)
\]

Subtypes$(t)$ is the set $\{t_i\}$ of children of $t$, such that $t_i \in N \cup Z$

Algorithm COMPLETE2:

Let $I_p = N$, $I = \{\}$;
For each $t_i \in I_p$ do
- $\text{Max}(t_i) = i$;
- $\text{Parents}(t_i) = \{\}$;
- $\text{Subtypes}(t_i) = \text{all } t_j \in N \cup Z$, such that $t_j \leftarrow t_i$
While $I_p \neq \phi$
  Let $l_c = \{\}$
  For every $t \in I_p$ do
    For each $t' \in l_c$ do
      If $\text{Parents}(t) \subseteq \text{Parents}(t')$ then
        add $t' \leftarrow t$ to $D'$
    For $i = \text{Max}(t) + 1$ to $n$ do
      if $E(t) \cap E(t_i) \neq \phi$ then
        create a new type $t_i$
        add $t_i$ to $l_c$
        add $t_i \leftarrow t$ and $t_i \leftarrow t$ to $D'$
        Let $\text{Parents}(t_i) = \text{Parents}(t) \cup \{i\}$
        For each $t_m \in \cap \text{Subtypes}(t_i)$ where $t_j \in \text{Parents}(t_i)$ do
          Add $t_m \leftarrow t_i$ to $D'$
        Let $\text{Max}(t_i) = i$.
  Let $I = I \cup l_c$
  Let $I_p = I_c$
CLOSE(top);
end COMPLETE2.

Consider $S^I = (N \cup I, B, \leftarrow^I, T, \bot, \phi)$.

Let $\leftarrow^I$ be defined as, $t_i \leftarrow^I t_j$

if $t_i \leftarrow t_j$, $t_i, t_j \in N$, or $t_i \leftarrow^I t_j$, $t_i, t_j \in N \cup I$.

Thus, $\leftarrow^I$ is closed on $N \cup I$. And for all $t_i, t_j \in N \cup I$, there exists an intersection type $t'$, such that $t' \in N \cup I$.

7. Theory and Correctness of the Types Implementation.
For $S'$ to be a complete type structure the intersection type $t'$ of $t_i$ and $t_2$ must represent the true intersection of $t_i$ and $t_2$.

1. $\phi(t')$ contains all $o \in \phi(t_i) \cap \phi(t_2)$, because for all $o$, there exists a type $t_o$ with $o \in \phi(t_o)$ and $t_o \leftarrow t_i, t_o \leftarrow t_2$ (by algorithm COMPLETE1). 
   $t_o \leftarrow t'$, by construction (algorithm COMPLETE2), thus $o \in \phi(t')$.

2. $\phi(t')$ contains nothing but $\phi(t_i) \cap \phi(t_2)$, because each $t_o \leftarrow t'$ is either a user defined type or an extra type. If $t_o$ is a user defined type then the relations $t_o \leftarrow t_i$ and $t_o \leftarrow t_2$ exist. Also $\phi(t_o) \subseteq \phi(t_i) \cap \phi(t_2)$. If $t_o$ is an extra type, then it is created because an object $o \in \phi(t_i) \cap \phi(t_2)$ exists and $o \in \phi(t_o)$.

Thus $S$ is a complete type structure such that for all $t_i, t_2 \in N'$ there exists $t' \in N'$ such that

$\phi(t') = \phi(t_i) \cap \phi(t_2)$.

**Termination:**

Algorithms COMPLETE1 and COMPLETE2 both iterate on the same basic loop condition, i.e., until $Z_p$ or $l_p$ is empty. The set $Z_p$ or $l_p$ for the next iteration is created in the ‘for’ loop. Max(t) is a strictly increasing function, hence it is guaranteed to be $\geq n$ in finite time. Then, no new types are created and $Z_c$ is empty. Thus the algorithm definitely terminates. When there is little intersection between the elements of $N$ the algorithm will terminate faster.

**Equivalent Structures.**

Consider two structures $S = (N, B, \leftarrow, T, \perp, \theta)$

---

*7. Theory and Correctness of the Types Implementation.*
\[ S' = (N', B, \leftarrow', T, \perp, \theta') \]

S and S' are equivalent with respect to any \( N' \), where

\[ N' \subseteq N \cap N', \text{ iff} \]

for all \( t \in N' \), for all \( o \in B : o \in \theta(t) \text{ iff } o \in \theta'(t) \).

Operations that retain equivalence between type structures:

1. Adding a new type.
2. Adding a link between types \( t_i \leftarrow t_j \) if for all \( o \in \theta(t_i), o \in \theta(t_j) \)
3. Adding a term \( T \) to a type \( t \) if for all \( o \in M(T), \text{ for all } t' : t \leftarrow t', o \in \theta(t') \)

In all the constructions we perform, we have used the above operations and hence we construct equivalent structures.
8. Time and Space Complexity Analysis.

In this chapter, we discuss the complexity of the modified unification algorithm, the numerical constraints solver and the typed unification mechanism.

For the numerical constraint solver, we use a linear algebra technique known as the QR decomposition method. This involves several different algorithms. The decomposition algorithm is $O(n^2m)$, where $n$ is the number of constraints involved in the decomposition and $m$ is the number of variables in these constraints. The incremental addition of a constraint to the decomposed matrix is achieved by applying Givens rotations and retaining the matrix in its upper triangular form. This algorithm is $O(nm)$. The solution of the system is obtained by backsolving, which is $O(nm)$. Hence, the time complexity of the numerical constraint solver is $O(n^2m)$.

As all computations are on matrices the space complexity is also $O(nm)$.

Adding type inheritance to Prolog involves two stages. The first stage modifies the type structure to the state where the condition of intersection holds. This is achieved using the complete1 algorithm of chapter 7. This creates a structure which has, in the worst case, $2^n$ types, where $n$ is the number of types defined by the user. Hence the complexity of the algorithm is $O(2^n)$ in the worst case. The complexity depends on the total number of types in the modified structure and the number of types created de-
pends on \(d\), the maximum number of types that have a common intersection. In the worst case, when an \(n\)-way intersection exists, \(d\) is equal to \(n\).

This worst case exponential behaviour should not cause too much worry for the following reasons. Types were introduced to take advantage of structured domains. If all \(n\) types have intersecting elements, then the domain is not very structured and using an untyped domain is probably the best programming approach. Also, this algorithm is a part of preprocessing and is a one time cost only.

The second stage is the creation of all intersection types (glbs) for all pairs of types. This stage has been moved to runtime to avoid creating glbs that will not be required.

For the typed unification mechanism, we consider the cost of unifying a pair of clauses. Let \(m\) be the number of unifications required. Typed unification consists of two parts, syntactic matching and finding the glb of the types of the terms to be unified. Syntactic matching corresponds to Robinson's unification algorithm which is known to be of exponential cost in the worst case and of \(O(m)\) in the average case.

There is a table to store the glbs of all pairs of types. When a glb is created, a new type number is allocated to it, and the type number is stored in the table. The bitsets associated with both types are 'anded' together and the result is made the bitset associated with the glb. Therefore if a glb already exists, it can be found in constant time. Otherwise, the number of 'and' operations is proportional to \(n\), and is in the worst case \(2^n\).
To show that storing the glb and also moving the second stage to runtime is justifiable, we find the expected number of glbs that need to be created as the number of unifications becomes very large. We are performing an average case analysis here.

Let each unification of literals consist of m arguments to be unified. Hence m glbs are required. Let the total number of glbs that algorithm complete2 creates be N.

We define,

\[ E(k) = \text{the expected number of new glbs needed to be created to unify} \]
\[ \text{the kth pair of literals.} \]

\[ \rho_a(i) = \text{the probability that the glb for the ith} \]
\[ \text{unification of terms in the kth unification of literals does} \]
\[ \text{not already exist.} \]

Therefore,

\[ E(k) = \sum_{i=1}^{m} \rho_a(i) \times 1 \]

We assume each glb has an equal probability of not being in the table. Also the m glb operations in each unification of clauses are distinct.

By induction we show that \( E(k) = m(1 - r)^{k-1} \),

where \( r = \frac{m}{N} \).

Base Case:

\[ E(1) = m \times 1 = m \], as for the first unification of clauses no glbs exist in the table. we assume that \( E(k) = m(1 - r)^{k-1} \).

By induction, we show that \( E(k + 1) = m(1 - r)^k \)

\[
E(k + 1) = m \frac{(N - (m + m(1 - r) + \ldots + m(1 - r)^{k-1}))}{N}
\]
\[
= m(1 - r(1 + (1 - r) + \ldots + (1 - r)^{k-1}))
\]
\[
= m \frac{(1 - r(1 - (1 - r)^k))}{(1 - (1 - r))}
\]
\[
= m(1 - 1 + (1 - r)^k)
\]
Thus, taking the limit of $E(k)$ as $k \to \infty$, 

\[
\lim_{k \to \infty} m(1 - r)^{k-1} = 0 \quad \text{as} \quad (1 - r) < 0
\]

Therefore, as the number of clauses we unify becomes larger the number of glbs that need to be created tends to zero.

As the largest structure we are constructing is $2^n$, the space complexity of this algorithm is $O(2^n)$.
9. Results.

Here we show some example runs of the two modules, the numerical constraint solver and the type inheritance mechanism. We have implemented them separately and the following examples illustrate the performance of the modules.

**Incorporating Arithmetic.**

**Example 1.**

`{* rules *}

   (= -6 [- [- ?x ?y] ?z])
   (Hint ?x ?y)
   (Positive ?y))

ii) ((Hint ?x ?y) if (= 4 [- ?y ?x]))

iii) ((Hint ?x ?y) if (= 2 [- ?y ?x]))

Query:

(Find ?x ?y ?z)

This will match with i) to give two equations

?x + ?y + ?z = 12

?x - ?y - ?z = -6

We Solve this to find the system is consistent.
From ii) we get the additional equation
\[?y - ?x = -4\]
This gives the solution
\[?x = 3, ?y = -1, ?z = 10\]
But \(?y\) should be positive so we backtrack.

From iii) we get the equation
\[?y - ?x = 2\]
This gives the solution
\[?x = 3, ?y = 5, ?z = 4.\]

Example 2.
Suppose you wish to find out how much of food1, food2 and food3 will make a snack of 610 calories with 405 units of protein and totally 65gms. The distribution of calories and proteins in the three foods is as follows.

<table>
<thead>
<tr>
<th></th>
<th>Calories/gm</th>
<th>Proteins/gm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Food1</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Food2</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Food3</td>
<td>20</td>
<td>3</td>
</tr>
</tbody>
</table>

{ /* rules */

i) (meal ?f1 ?f2 ?f3) if

\[(= 610 [+ [+ [* 8 ?f1] [* 6 ?f2]] [* 20 ?f3]])\]
\[(= 405 [+ [+ [* 5 ?f1] [* 2 ?f2]] [* 3 ?f3]])\]

Query:

(meal ?x ?y ?z)

9. Results.
From i) we get the set of equations

\[ 5 \cdot \varphi_1 + 8 \cdot \varphi_2 + 20 \cdot \varphi_3 = 610 \]
\[ 8 \cdot \varphi_1 + 6 \cdot \varphi_2 + 3 \cdot \varphi_3 = 405 \]
\[ \varphi_1 + \varphi_2 + \varphi_3 = 65 \]

This when solved gives the solution

\[ \varphi_1 = 50, \ \varphi_2 = 160, \ \varphi_3 = 100. \]

Type Inheritance.

The syntax used below is described in the Appendix.

Example 1.

The program segment consists of the following declarations and rules. See Figure 3 (end of chapter) for the hierarchy. {\* declarations *}

\( \text{(animals < livingthings)} \)
\( \text{(plants < livingthings)} \)
\( \text{(carnivores < livingthings)} \)
\( \text{(domestic < carnivores)} \)
\( \text{(wild < carnivores)} \)
\( \text{(domestic < animals)} \)
\( \text{(wild < animals)} \)
\( \text{(venusflytrap < < carnivores)} \)
\( \text{(venusflytrap < < plants)} \)
\( \text{(pitcherplant < < carnivores)} \)
(pitcherplant << plants)
(dog << domestic)
(cat << domestic) {* rules *}

i) ((Eatsmeat ?x: carnivore))

ii) ((Chasesmailman ?x:domestic) if Barks(?x))

iii) ((Barks Dog))

Queries:

1. (Eatsmeat ?x: plant)
   returns (Eatsmeat ?x: anonymous),
   {* where anonymous is an intersection node that holds the two objects
   venustrap and pitcherplant *}

2. (Eatsmeat ?x: animal)
   returns (Eatsmeat ?x: domestic | wild)
   {* domestic | wild is an intersection node that is created *}

3. (Chasesmailman ?x: domestic)
   returns (Chasesmailman Dog)

Example 2.
This program segment consists of the following declarations and rules. See Figure 4 (end of chapter) for the hierarchy.
   {* declarations *}
   (birds < has_lungs)

9. Results.
(birds < lays_eggs)
(reptiles < has_lungs)
(reptiles < lays_eggs)
(owls < birds)
(parrots < birds)
(snakes < reptiles)
(polly < < parrots)
(kaa < < snakes)

{" rules *

i) ((animal ?x: lays_eggs) if (warmblooded ?x))
ii) ((animal ?x: lays_eggs) if (coldblooded ?x))
iii) ((warmblooded ?y: birds))
iv) ((coldblooded ?y: reptiles))
v) ((talks polly))

Queries:

1. (animal ?x: has_lungs)
   unifies with i) to give (animal ?x: birds | reptiles)
   unifies with iii) to give (warmblooded ?x: birds).
   returns ((animal ?x: birds))

2. (warmblooded ?x: lays_eggs)
   returns ((warmblooded ?x: birds))

3. (talks ?x: birds)
   returns ((talks polly))

9. Results.
Example 3.

Consider the following program segment. See Figure 5 (end of chapter) for the hierarchy.

{ * declarations * }
(undergraduates < university)
(graduates < university)
(csdept < university)
(mathdept < university)
(cs < < dept)
(math < < dept)
([[student ?x ?y] < < university])
([[student john cs] < < undergraduate])
([[student mark math] < < undergraduate])
([[student ?x cs] < < csdept])
([[student jane cs] < < graduate])
([[student tom cs] < < graduate])

{ * rules * }
((takes Al ?x: csdept))
((enrolled ?x: university))
((uses_gym [student ?x ?y: dept]: university))

Queries:

1. (takes Al [student jane cs])
   returns (takes Al [student jane cs]: csdept).

2. (enrolled [student john cs])

9. Results.
returns (enrolled [student john cs]: university)

3. (uses_gym [student ?x math])

    returns (uses-gym [student ?x math]: university)
Figure 3. Type Inheritance: Example 1.
Figure 4. Type Inheritance: Example 2.
Figure 5. Type Inheritance: Example 3.
10. Conclusions and Future Work.

We have created a Prolog compiler incorporating function evaluation and type inheritance into unification. This was achieved using the constraint satisfaction model of computation. We spoke of the triangle with unification, incorporating arithmetic and inheritance as the vertices, we succeeded in integrating unification and function evaluation, and unification and inheritance.

Our concept of a typed Prolog is an extension of the WAM concept. The representation of terms and the data structures used are the same as those for standard Prolog. We have also incorporated arithmetic in this Prolog compiler. We have also showed the soundness and termination properties of our type implementation.

We discuss a comparison of this work with LOGIN [AITK 86] to emphasize the advantages of our implementation.

Comparison with LOGIN.

In LOGIN, terms are represented using record structures. A term consists of a type symbol and a list of attributes with their labels. All attributes are basically terms and hence are represented by records. Since terms can have different numbers of arguments, the labels are required, and the labels are also stored in the structure. Even
normal Prolog terms must use the same record structure representation. In our implementation we have a single 32 bit representation for each term. This is the same representation introduced by Warren for the WAM.

Backtracking cannot be a simple process using the LOGIN kind of data structure. There are fields in the record representation of the term to store the changed type and attributes that result during unification. This cannot take care of cases when a series of unifications on the same term occur and backtracking to an intermediate stage is required. Backtracking in our implementation consists of storing the address of a term that is to be trailed together with the actual contents of the term.

LOGIN is an entirely new logic programming language with types and inheritance and an entirely new implementation. It can handle untyped logic programs equivalent to Prolog. Our implementation is an extension of Prolog, and hence untyped Prolog programs will still run on our system just as before typing was added.

Inheritance is a key property required for object-oriented languages. By adding inheritance to Prolog we have added a link between logic programming and the object oriented paradigm.

The type of constraints handled by our system can be extended. To do this we would need to store the constraints in a different data structure. For each constraint, the type of the constraint needs to be stored, and the constraint itself must be stored as a relation.

Non-linear arithmetic constraints can be handled using the delay technique. That is, solving non-linear constraints is delayed in the hope that in the future some variables
get instantiated and the constraint becomes linear. A generic method to store the constraint will handle the representation of the constraint. To check the linearity of the constraint, we use an expression simplifier that returns an error message if the constraint is non-linear. The same procedure can be applied to the non-linear constraint to check whether it has become linear after some variables are instantiated. Then the new system of equations would be solved using the QR decomposition method.

Constraints of the inequality type could be handled using linear programming techniques such as the simplex method.


[XU 88] J.Xu, D.S.Warren *A Type Inference System for Prolog*


Here we describe an extension of Prolog that adds types with inheritance to HC-Prolog. Standard HC-Prolog programs will run on this system. It is necessary to declare the hierarchy of types and the elements of types before any rules are defined.

We define a few words before discussing the syntax.

**Atom**  
Objects in the universe are called atoms. For example, dog, '+', [student john cs]. The latter is called a functor object.

**Term**  
All standard Prolog terms are terms. For example, ?x, (?x (a nil)). All atoms are terms. variable: atom, [student ?x: name cs]: university (where name, atom and university are type names), are terms.

The syntax for our extended Prolog is as follows:

1. (atom1 < atom2), where atom1 and atom2 are type names. This declares that atom1 is a subtype of atom2.

2. (term << atom1), where atom1 is a type name. This declares that the set of objects that unify with term (if term is an atom, a single object), are elements of type atom.
Example 1.

The program segment consists of the following declarations and rules. {* declarations *}

(animals < livingthings)
(plants < livingthings)
(carnivores < livingthings)
(domestic < carnivores)
(wild < carnivores)
(domestic < animals)
(wild < animals)
(venusflytrap < < carnivores)
(venusflytrap < < plants)
(pitcherplant < < carnivores)
(pitcherplant < < plants)
(dog < < domestic)
(cat < < domestic) {* rules *}

(assert

((Eatsmeat ?x: carnivore)) ;(i)
((Chasesmailman ?x:domestic) if Barks(?x)) ;(ii)
((Barks Dog)) ;(iii)
)

Queries:

1. (Eatsmeat ?x: plant)

returns (Eatsmeat ?x: anonymous).
2. (Eatsmeat ?x: animal)
   returns (Eatsmeat ?x: domestic | wild)
   {* domestic | wild is an intersection node that is created *}

3. (Chasesmailman ?x: domestic)
   returns (Chasesmailman Dog)

Example 2.
This program segment consists of the following declarations and rules.

{* declarations *}
(birds < has_lungs)
(birds < lays_eggs)
(reptiles < has_lungs)
(reptiles < lays_eggs)
(owls < birds)
(parrots < birds)
(snakes < reptiles)
(polly < < parrots)
(kaa < < snakes)

{* rules *}
(assert
  ((animal ?x: lays_eggs) if (warmblooded ?x)) ;(i)
  ((animal ?x: lays_eggs) if (coldblooded ?x)) ;(ii)
(warmblooded ?y: birds) ;(iii)
(coldblooded ?y: reptiles) ;(iv)
(talks polly)) ;(v)

Queries:

1. (animal ?x: has_lungs)
   unifies with i) to give (animal ?x: birds | reptiles)
   unifies with iii) to give (warmblooded ?x: birds).
   returns ((animal ?x: birds))

2. (warmblooded ?x: lays_eggs)
   returns ((warmblooded ?x: birds))

3. (talks ?x: birds)
   returns ((talks polly))

Example 3.

Consider the following program segment.

{"declarations *}

(undergraduates < university)
(graduates < university)
(csdept < university)
(mathdept < university)
(cs << dept)
(math << dept)
([student ?x ?y] <<< university)
([student john cs] <<< undergraduate)
([student mark math] <<< undergraduate)
([student ?x cs] <<< csdept)
([student jane cs] <<< graduate)
([student tom cs] <<< graduate)

{" rules *}

(assert

((takes AI ?x: csdept))
((enrolled ?x: university)
  ((uses_gym [student ?x ?y: dept]: university))))
)

Queries:

1. (takes AI [student jane cs])
   returns (takes AI [student jane cs]: csdept).

2. (enrolled [student john cs])
   returns (enrolled [student john cs]: university)

3. (uses_gym [student ?x math])
   returns (uses-gym [student ?x math]: university)
Appendix B. Extending the Incorporation of Arithmetic into Unification.

Constraints need to be stored in a more general form, i.e., $(R, \text{term}, \text{term})$, where $R$ is the relation or constraint between the two terms.

In the present system, each numerical constraint is simplified and stored in its simplified form. The simplified form is basically a list of coefficient-variable pairs, with only one term for each variable. The constant is stored first with a null variable. The constraint is converted so that its right-hand side is zero. The simplifier checks for linearity of the constraint by simplifying the constraint into the sum of coefficient-variable pairs. If this is not possible it returns false otherwise it returns a list of the coefficient-variable pairs. The coefficient matrix is created just before QR decomposition is applied to the system. To handle non-linear constraints all constraints must be stored in the general form, so that the non-linear constraint is tested for linearity before the coefficient matrix is created. The non-linear constraints will remain in general form until they become linear. When the result of solving a system gives a unique solution, the non-linear constraint can be re-tested for linearity.
Appendix C. Changes to the WAM to incorporate Type Inheritance.

Terms can still be represented in the 32 bit representation with a tag that specifies the type of the term. The tag is in the upper 8 bits of the term. We added the type `\texttt{functerm}' to the already existing term types. The functerm points to a structure that is a functor term, i.e., a functor symbol followed by the type of the functor symbol and arguments. The number of the arguments for a functor term are fixed and that number is stored in the atom hash table associated with the functor symbol.

Variables and functor terms can be typed. In the 32 bits that represent a variable, a tag in the first 8 bits and the type number of the variable in 16 bits are packed together. The functor term has the type stored as a separate field together with the functor symbol and the arguments of the functor in the structure. The structure of a functor term can be represented in \((n + 2) \times 32\) bits, where \(n\) is the number of arguments, and the other 2 terms are for the functor symbol and the type.

To include type inheritance, we need to add a few instructions to the existing WAM instruction set.
Get Instructions.

GetTypedTVar $X, A, S1$

This instruction represents a head argument that is a typed temporary variable, where $S1$ is the type of the variable. The instruction gets the type $S2$ of $A$, trails $A$, if not ($S2 < S1$), and sets its type to the $\text{GLB}(S1, S2)$. Then $A$ is stored in $X$.

GetTypedPVar $V, A, S1$

This instruction represents a head argument that is a typed permanent variable, where $S1$ is the type of the variable. The instruction gets the type $S2$ of $A$, trails $A$, if not ($S2 < S1$), and sets its type to the $\text{GLB}(S1, S2)$. Then $A$ is stored in $V$.

GetStructure $F, A, S1$ (* $S1 = 0$, if no type *)

This instruction is used when the head argument is a functor term with type $S1$. The value of register $A$ is dereferenced.

If the result is a reference to a variable with type $S2$ then the variable is bound to a new structure pointer at the top of the heap and execution proceeds in "write mode". The type of the structure is set to $\text{GLB}(S1, S2)$.

Otherwise, if the result is a structure of type $S2$, and its functor symbol is identical to functor $F$, the structure pointer $S$ is set to point to the arguments of the structure, and execution begins in "read mode". The type of the structure is set to $\text{GLB}(S1, S2)$.
Put Instructions.

PutTypedTVar $X_i, A_\nu, S1$
This instruction represents a goal argument that is a typed temporary variable, where $S1$ is the type of the variable. The instruction creates an unbound variable on the heap with the type $S1$, and puts a reference to the heap term into registers $A_\nu$.

PutTypedPVar $V_i, A_\nu, S1$
This instruction represents a goal argument that is a typed permanent variable, where $S1$ is the type of the variable. The instruction creates an unbound variable $V_i$ with type $S1$ and puts a reference to $V_i$ in $A_\nu$.

PutStructure $F, A_\nu, S1$ (* $S1 = 0$, if no type *)
This instruction is used when the goal argument is a functor term with type $S1$. The instruction pushes the functor symbol on the heap, sets the type to $S1$, and puts a corresponding structure pointer into register $A_\nu$. Execution continues in "write mode".

Unify Instructions.

UniTypedTVar $A_\nu, S1$
This instruction is used when the argument is a temporary variable of type $S1$.

- In write mode, it creates an unbound variable at $S$ and sets its type to $S1$. $A_\nu$ is made a reference to $S$.  

Appendix C. Changes to the WAM to incorporate Type Inheritance.  

96
• In read mode, it trails the term pointed at by S if not (S2 < S1), sets its type to GLB(S1,S2), where S2 was its type before setting. A, is made a reference to S.

UniTypedPVar V,, S1
This instruction is used when the argument is a permanent variable with type S1.

• In write mode, it creates an unbound variable at S, and sets its type to S1. V, is made a reference to S.

• In read mode, it trails the term pointed at by S if not (S2 < S1), sets its type to GLB(S1,S2), where S2 was its type before setting. V, is made a reference to S.

An example to show the code generated.

The program segment consists of the following declarations and rules. {* declarations *}

(animals < livingthings)
(plants < livingthings)
(carnivores < livingthings)
(domestic < carnivores)
(wild < carnivores)
(domestic < animals)
(wild < animals)
(venusflytrap << carnivores)
(venusflytrap << plants)
(pitcherplant << carnivores)
(pitcherplant << plants)
(dog << domestic)
(cat << domestic) {* rules *}
i) ((Eatsmeat ?x: carnivore))
ii) ((Chasesmailman ?x: domestic) if Barks(?x))
iii) ((Barks Dog))

Queries:

1. (Eatsmeat ?x: plant)
   returns (Eatsmeat ?x: anonymous),
   {* where anonymous is an intersection node that holds the two objects
    venustrap and pitcherplant *}

2. (Eatsmeat ?x: animal)
   returns (Eatsmeat ?x: domestic | wild)
   {* domestic | wild is an intersection node that is created *}

3. (Chasesmailman ?x: domestic)
   returns (Chasesmailman Dog)

The type numbers corresponding to the declared types are as follows:

1 - Animals
2 - LivingThings
3 - Plants
4 - Carnivores
5 - Domestic

Appendix C. Changes to the WAM to incorporate Type Inheritance.
The code generated (using [WARR 83] and the new instructions) for the three rules and queries are as follows.

EatsMeat: getTypedTVar R2 from R1, 4
  return

ChaisesMailMan: getTypedTVar R2 from R1, 5
  putTVar R2 into R1
  goto Barks

Barks: getCon atom 'Dog' from R1
  return

Query1: putTypedPVar V0 into R1, 3
  call Eatsmeat
  halt

Query2: putTypedPVar R2 into R1, 1
  call Eatsmeat
  halt

Query3: putTypedPVar R2 into R1, 5
  call ChaisesMailMan
  halt.

Appendix C. Changes to the WAM to incorporate Type Inheritance.
The vita has been removed from the scanned document