Hierarchical Constraint Satisfaction as a Model for Adding Types with Inheritance to Prolog

By Chandan Chitale, John Deighan and John Roach

TR 90-6

Technical Report SRC-90-003

Hierarchical Constraint Satisfaction as a Model for Adding Types with Inheritance to Prolog*

Chandan Chitale, John Deighan, and John Roach

Department of Computer Science
at
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

May 1989

*Work supported by the U.S. Navy through the Systems Research Center under Basic Ordering Agreement N60921-83-G-A165 B044.

*Cross referenced as CS-90-6, Department of Computer Science, Virginia Tech.

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. Prolog operates on the Herbrand universe, a single unstructured domain. In problems with large structured domains, the number of resolution steps required may become large. We have incorporated type inheritance into Prolog to exploit large structured domains to write more concise code and to obtain shorter proofs. Types are subuniverses corresponding to sets of objects. The subset of relation between types induces a hierarchy on the universe. We use hierarchical constraint satisfaction concept to incorporate these extensions into Prolog. We also provide a formal proof that our typed unification extends standard Prolog and directly augments the Warren Abstract Machine (WAM) concept.

CR Categories and Subject Descriptors: I.2.3 [Artificial Intelligence]: Deduction and Theorem Proving - *Logic programming*

Additional Key Words and Phrases: Inheritance, types, Prolog

1.Introduction.

Prolog, a programming language based on a subset of first order logic, has generated considerable interest in recent years because of its simplicity, its use as an artificial intelligence language, and its clean semantics. The heart of the language is unification, the procedure that performs matching; each computation step, that is, each inference, requires a call to unification. Any extension or improvement to the unification procedure therefore extends the power and affects the execution speed of the Prolog language. Many research efforts [COLM 84] [AITK 86] [ROAC 89] are consequently exploring improved unification algorithms.

Object oriented programming languages consist of inheritance, encapsulation and message passing mechanisms. Combining some of the best features of object oriented programming with Prolog would create a language of great expressive power. In this paper, we extend the unification algorithm to incorporate inheritance. With the extensions we provide, the programmer may, if desired, create a type hierarchy for domain objects enabling an object to inherit type information from objects above it in the hierarchy.

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

```
((animal ?x) if (bird ?x))
((animal ?x) if (mammal ?x))
((mammal ?x) if (whale ?x))
((whale mobydick1))
.....
((whale mobydick20))
((breathes ?y  air) if (animal ?y))
```

and consider the query ((breathes mobydick20 air)).

Solving this query requires 4 resolution steps and several backtracking steps to determine that the goal is true.

With the modified unification algorithm we report here, the programmer may subdivide the domain of discourse into several types, or subuniverses. These subuniverses could be disjoint, or intersecting, or subsets of each other. Using the syntax of our system, we re-implement the above program as follows.

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

Now, the same query results in a true value in a single resolution step without 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.

In this paper, we first review the relevant literature and then introduce constraint satisfaction as a model of computation appropriate for extending the unification algorithm. Next we present an implementation of the computational model and then provide a formalization and proof of correctness for it. We conclude with an example that illustrates code our system can execute and discuss the relative merits of our system. An appendix details changes to the Warren Abstract Machine (WAM) needed to extend current technology Prolog compilers.

2.Literature Review.

Many-Sorted Unification.

In [WALT 88], the universe is divided into subsets or sorts and is defined as a potentially infinite hierachy 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 hierachy 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 hierachy 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 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.

A declaration of inheritance information is called a 'signature'. The type signature Σ 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 Σ 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 ψ term. A ψ term consists of a root symbol, attribute labels with associated sub-ψ terms and coreference constraints among paths of labels. Conventional first-order terms are a particular instance of the ψ term. ψ terms are treated like records with attributes as fields. By introducing attributes to ψ 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 ψ 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 ψ term unification algorithm. This algorithm computes the ψ. term which is the greatest lower bound of the two given ψ terms. The tags for attribute labels are used to build coreference classes that have class representatives. All tags in the unified ψ term are replaced by their class representatives, thus resolving coreference. Σ unification returns $\perp$ (if the ψ terms cannot unify) or the ψ term built after resolving coreference.

The sorts structure needs to be a lower semi- lattice for this algorithm to work, i.e., 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 $2^{(s)}$ of the partially-ordered set $(S, \leq)$ of sort symbols. The set $2^{(s)}$ 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 ψ 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 glb operation is the most frequently used operation during unification, but nothing is mentioned as to how it is achieved. The 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 ψ term concept has moved LOGIN away from standard Prolog syntax and semantics that are incorporated in the WAM [WARR 83]. We compare and contrast our approach with LOGIN in the discussion and conclusions section at the end of the paper.

3.Treating Inheritance as Hierarchical Constraint Satisfaction.

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 'subset 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 constraints locally and then propagating the results globally. A constraint satisfaction problem consists of variables, each with an associated domain and a set of constraining 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 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 $D_i \subseteq V$.

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

   $\{A | A \in D_1 \times D_2 \times ... \times D_k\}$.

6. Solution criterion:

   An assignment A is a solution iff

   $\forall i, j \leq k$, if $i \neq j$ then

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

```
procedure REVISE-A ((i,j))
begin
  Δ ← {x|(x∈Δi)∧[(∃y)(y∈Δj)∧Rij(x,y)]}
  DELETE ← (Δ ⊂ Δi)
  if DELETE then Δi←Δ
    return DELETE
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.

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 $\{D_i^{qs}\}$, 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].

```
procedure REVISE-HAC ((i,j));
begin
  DELETE ← false;
  Q1 ← Δi
  Δi ← φ
  while Q1 not empty do
  begin
    select and delete an element Di^qs from Q1
    Q2 ← Δj
    FOUND ← false
    while Q2 not empty and not FOUND do
    begin
      select and delete an element Dj^lt from Q2
      if ∀ elements of Di^qs exists an element of Dj^lt that satisfies Rij then
         begin
           Δi←Δi∪{Di^qs}
           FOUND ← true
         end
     end
     if not FOUND then
     begin
       DELETE ← true
       if q > 0 then
       begin
         Q2←Δj
         while Q2 not empty and not FOUND do
         begin
           select and delete an element Dj^lt from Q2
           if for some element of Di^qs ∃ an element of Dj^lt that satisfies Rij then
           begin
             Q1←Q1 ∪ { subsets of Di^qs }
             FOUND ← true
           end
         end
       end
    end
  end
 return DELETE
end.
```

It has been claimed that these algorithms are correct and do terminate.

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 Δ 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 section we discuss an algorithm that combines the two ideas: symbolic unification and inheritance.

4. Treating Unification as Constraint Satisfaction.

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

- either $s_i \sigma = t_i \sigma$
- or $S' = S \cup \{ < =, s_i \sigma, t_i \sigma > \}$ [ROAC 89] discusses an algorithm, "constrained unification" that treats unification as a constraint satisfaction problem thereby incorporating function evaluation into the normal unification procedure.

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 $T_1,...,.T_n$ are typed terms, then [f $T_1,....,T_n$]: 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.

Typed or Hierarchically Constrained Unification Algorithm.

We modify the constrained unification algorithm to handle unification of typed terms. Constraints are relations between pairs of terms. In this system, the type of constraints that hold are 'subset of', 'element of' and 'equality'. The first two kinds of constraints are created during the declaration stage of the program and already exist in S. If there are no declarations S will be empty initially. The 'equality' constraint is created during unification at runtime.

<u>Algorithm.</u>

Let T = $\{(s_1, t_1),(s_2, t_2).....\}$ be the set of pairs of terms to be unified. Let σ be the set of substitutions. Set σ to ϕ. Let S be the set of constraints. Initially S may or may not be empty.

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 σ 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 a variable. Then if s_i is an element of the type of t_i then add $t_i = s_i$ to σ and goto 1.

   c. t_i is a functor term, then stop because unification has failed.

5. If s_i is a variable and
   a. t_i is a variable. Then let t' be the glb of the types of s_i and t_i. If $t' \neq$ **Bottom** then set the type of t_i to t',add $s_i = t_i$ to σ, else stop because unification has failed.
   b. t_i is a functor term. Then let t' be the glb of the types of t_i and s_i . If $t' \neq$ **Bottom** then let the type of t_i be t'. Add $s_i = t_i$ to σ. Else stop because unification has failed.
6. If s_i is a functor term and
   a. t_i is also a functor term. Then let t' be the glb of the types of t_i and s_i . If $t' \neq$ **Bottom,** set the type of s_i to t'. Add $t_i = s_i$ to σ and goto 1. Else stop because unification has failed.

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 $<=, t_1, t_2>$ is added to S. Consider the set M of all constraints that t_1 and t_2 occur in. We remove a single constraint from M. We prune the domains of each variable in the constraint we re-

moved, by removing all values for which no values exist in the other variables for the constraint to be consistent.

5.Implementation.

In Prolog, terms are defined over a single universe, 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. Since subuniverses may be contained in one another, the 'subset of' relation induces a partial ordering on types. A type t_1 that is a 'subset of' t_2 inherits all the properties of t_2.

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, hierarchically constrained unification, 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 ?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_1 is reachable from t_2 by more than one link, then the objects of t_1 are said to be indirect elements of t_2.

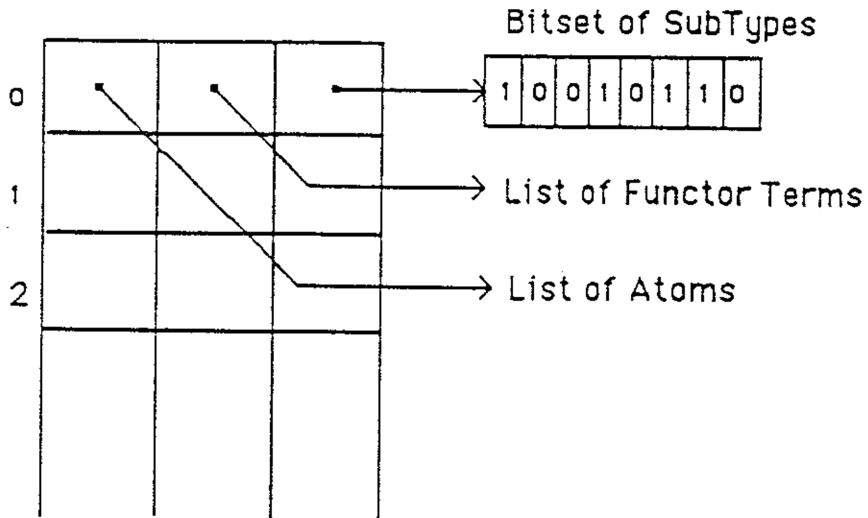
The type structure explained above is a directed graph consisting of types as nodes. There exists an arc from t_1 to t_2 if t_2 is a subtype of t_1. The graph must not have any cycles or loops, and user input is validated to avoid this.

The type structure is stored in a table (see Figure 1) 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 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.



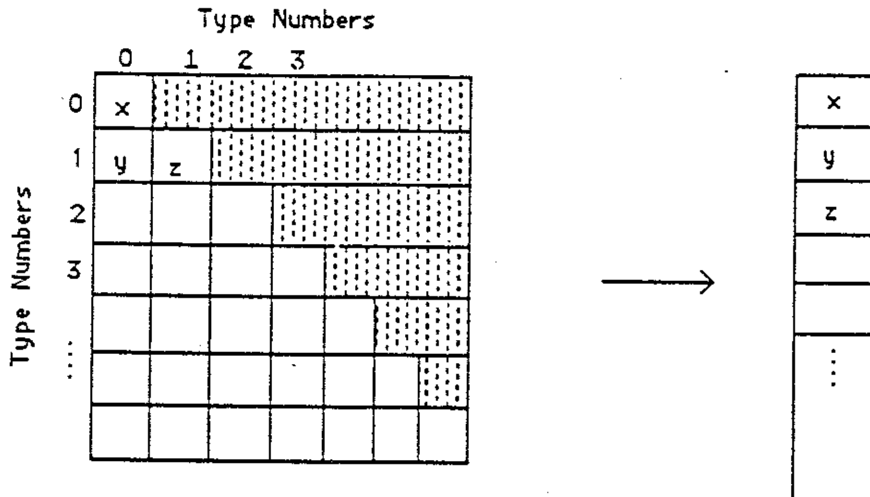


Figure 1. Data Structures for Adding Types with Inheritance to Unification.

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 1). 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.

Our implementation can be seen as a direct extension of the WAM [WARR 83] because we have used the same basic data structures and representation of terms. In an appendix, we describe the modifications made to the WAM and also the new instructions to be added to the code generated by the WAM.

6. Theory and Correctness of the Types Implementation.

Objects.

The universe consists of objects. Objects are 'things' and can be represented by **atoms**, e.g. cat, and **functors objects**, e.g [student joe 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 ,.... T_n are terms, then [f T_1 .... 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 ,.... 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 ,.... 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),......} , 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 to mean (s_i, t_i) $\in$ P.

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: term $\rightarrow$ set of objects, then

1. M(variable) = H, where H denotes the set of all objects in the universe.

2. M(object) = {object}

3. $M(F) = \{ V_1(F), V_2(F),....\}$, where $V_1, V_2,...$ 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,

$M(R) = \cup_{(T \in R)} M(T)$.

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 $\{ T_1 ,.... T_n \}$ to a term T' such that

$M(T') = M(T_1) \cap ... \cap M(T_n)$.

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, $G = (N, \leq, T, \perp)$, 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 ($t_1 \leq t_3$), ie.,transitivity holds.

2. for all t_1 , $t_2 \in N$: ($t_1 \leq t_2$ and $t_2 \leq t_1$) implies ($t_1 = t_2$), ie. t_1 and t_2 are the same type and no cycles exist in the graph.

Type Structure.

A type structure $S = (N, B, \leq, T, \perp, \theta)$ is an ordered 6-tuple, where $(N, \leq, T, \perp)$ forms a closed type graph, B is a set of objects, and θ 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 $S = (N, B, \leq, T, \perp, \theta)$ is a type structure for which, if o is an object and t_1 and t_2 are types , then

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

Complete Type Structure.

A complete type structure $S = (N, B, \leq, T, \perp, \theta)$ 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 $\{0,1,\ldots 2^{|N|}\}$.

2. $\leq'$ is defined as

   $t_s \leq' t_u$ iff $u \subseteq s$

3. $T' = t_\phi$

4. $\bot' = t_{\{1 \ldots n\}}$

5. θ' is defined as

   $\theta'(t_s) =$ B, if $s = \phi$

   else $\bigcap_{(i \in s)} \theta(t_i)$

To show that S' is a complete type structure,

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

1. G' is a simple type graph, because

   $T' = t_\phi \in N'$

   $\bot' = t_{\{1..n\}} \in N'$

   For all $t_s \in N'$: $\bot' = t_{\{1..n\}} \leq' t_s$ because $s \subseteq \{1..n\}$

   For all $t_s \in N'$: $t_s \leq' T' = t_\phi$ because $\phi \subseteq s$

   For all $t_s \in N'$: $t_s \leq' t_s$ because $s \subseteq s$

2. G' is a closed type graph because

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

$= (u \subseteq s)$ and $(v \subseteq u)$

which implies $v \subseteq s$

$= t_s \leq' t_v$

For all $t_s, t_u \in N'$:

$(t_s \leq' t_u)$ and $(t_u \leq' t_s)$

$= (u \subseteq s)$ and $(s \subseteq u)$

implies $u = s$, and therefore $t_u = t_s$

3. S' is a closed type structure because

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

$= o \in \bigcap_{(i \in s)} \theta' (t_i)$ and $u \subseteq s$,

but, $\bigcap_{(i \in s)} \theta' (t_i) \subseteq \bigcap_{(i \in u)} \theta'(t_i)$

thus $o \in \bigcap_{(i \in u)} \theta' (t_i)$

$= o \in \theta'(t_u)$

4. For any two types t_s and t_r, we prove that $t_{(s \cup r)}$ is their intersection type:

i.e. $\theta'(t_{(s \cup r)}) = \theta' (t_s) \cap \theta'(t_r)$

$\bigcap_{(i \in (s \cup r))} \theta (t_i) = (\bigcap_{(i \in s)} \theta(t_i) \cap (\bigcap_{(i \in r)} \theta (t_i))$

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, $\perp$, θ), 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 $T_1,\ldots T_n$ are typed terms, then ([f T_1 , ... T_n]: t) is a typed term
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

$$M(\text{T:t}) = M(\text{T}) \cap \theta(\text{t})$$

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 = $\{v_1{:}t_1/T_1,\ldots, v_n{:}t_n/T_n\}$, 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 $M(T_i) \subseteq \theta(t_i)$. 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(variable:t) = θ(t)

2. M(object:t) = {object} if object $\in \theta$(t),

otherwise ϕ

3. M(F) = $\{V_1(F), V_2(F), \ldots\}$, where $V_1, V_2, \ldots$ are all **permissable 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 $T_1, \ldots T_n$ are typed terms, then [f $T_1 \ldots T_n$] 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 $T_1{:}t_1$ and $T_2{:}t_2$ be two terms. If there exists $T'{:}t'$ such that

$$M(T'{:}t') = M(T_1{:}t_1) \cap M(T_2{:}t_2) \qquad (1)$$

then $T'{:}t'$ is the unifier of $T_1{:}t_1$ and $T_2{:}t_2$

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

$$M(T') \cap \theta(t') = (M(T_1) \cap \theta(t_1)) \cap (M(T_2) \cap \theta(t_2))$$
$$= (M(T_1) \cap M(T_2)) \cap (\theta(t_1) \cap \theta(t_2))$$

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:

$$\theta(\mathrm{t}) = \theta(t_1) \cap \theta(t_2)$$

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 $t_2 \in \mathrm{D}(t_1)$. We say that t_2 is directly reachable

from t_1 if $t_2 \leftarrow t_1$ and that t_n is indirectly reachable from t_1 if $t_2 \leftarrow t_1$ and t_n is indirectly reachable from t_2. D must include

- $t_i \leftarrow$ **top**, **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 $N \rightarrow 2^R$. E maps a type to a set of terms representing the objects belonging to that type. We introduce the symbol '$\propto$' and let T $\propto$ 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,

$D' : N \rightarrow 2^N$ is defined by,

$t_i \in D'(t)$ iff $t_i \in D(t)$ or ($t_i \in D(t_j)$ 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_i) then

CLOSE(t_i);

end;

$D'(t) = D(t) \cup D(t_i)$

end CLOSE.

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

It is trivial to show that G = (N, $\leftarrow$, **top**, **bottom**) is a simple type graph.

Now G = (N, $\leftarrow'$, **top**, **bottom**) is a closed type graph, where

$t_i \leftarrow' t_j$ means $t_i \in D'(t_j)$ because

1. For all t_1, t_2, t_3,

   if $t_i \leftarrow' t_2$ and

   $t_2 \leftarrow' t_3$ then

   $t_1 \in D'(t_2)$ and $t_2 \in D'(t_3)$. By the construction of D' , $t_1 \in D'(t_3)$

   Therefore, $t_1 \leftarrow' t_3$, i.e. transitivity holds.

Constructing a closed type structure.

Given a declared type structure L = (N, R, D, E), where G = (N, $\leftarrow$, **top**, **bottom**) is a closed type graph, we construct a closed type structure, S = (N, B, $\leftarrow$, **top**, **bottom**, ϕ), where

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

We define $\propto^*$ such that, T $\propto^*$ t_i

iff T $\propto$ t_i or ((T $\propto$ t_j) and ($t_j \leftarrow t_i$))

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

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

ϕ(t) = $M(E'$(t)).

S is a closed type structure because:

1. (N, $\leftarrow$, **top**, **bottom**) is a closed type graph.

2. ϕ is a mapping from $N \rightarrow 2^B$

3. o $\in \phi(t_1)$ means o $\in M(E'(t_1))$

but since $t_1 \leftarrow' t_2$ then $E'(t_1) \subseteq E'(t_2)$ and therefore $M(E'(t_1)) \subseteq M(E'(t_2))$, this implies,

$$o \in M(E'(t_2))$$

that is o $\in \phi(t_2)$

Constructing a complete type structure.

Given a declared type structure L = (N,R,D,E) where S = (N, B, $\leftarrow$, **top**, **bottom**, θ), is a closed type structure, we need to construct a complete type structure S' = (N $\cup$ I, B, $\leftarrow'$, **top**, **bottom**, ϕ'). 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_i such that ($T_i \propto t_i$ or $T_i \propto^* t_i$) with each term T_j such that ($T_j \propto t_j$ or $T_j \propto^* t_j$).

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^1, which includes pairs containing members of Z. In the second stage, another set of types I is created, which represent the true 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 $Z_1, \ldots Z_n$, where n = |N| and Z = $Z_1 \cup \ldots Z_n$.(note: If Z_i = { }, where i < n, then each of $Z_{(i+1)}, \ldots Z_n$ will be empty). To create the set Z_k, we need sets N and Z_{k-1} .

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

Each $t_c \in Z$, 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_i ∈ Z_p do
   Max(t_i) = i;
While Z_p ≠ φ
   Let Z_c = { }
   For every t ∈ Z_p do
      For i = Max(t) + 1  to n do
        if not(t_i ← t) and not(t ← t_i ) then
           create a new type t_k
           for each T_i ∝ t_i and T_j ∝ t such that T_i and T_j unify
              let unify(T_i,T_j)  ∝ t_k
              add t_k to Z_c
              add t_k ←' t_i and t_k ←' t to D
              let Max(t_k) = i.
   Let Z = Z ∪ Z_c
   Let Z_p = Z_c
CLOSE(top);
```

end **COMPLETE1.**

Let D^1 denote the mapping D after the algorithm terminates, and $t_1 \leftarrow' t_2$ mean $t_2 \in D^1(t_1)$.

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 $t \in S$, $t' \leftarrow' t$, $T \propto t'$, 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 = $\{t_i,...t_j,t_k\}$.

Then, for $S' = \{t_i,...t_j\}$ the condition holds. Consider an object $o \in \cap_{(t \in S)} M(E(t))$. Let t^1 be the type such that $o \in M(t^1)$ and $t^1 \leftarrow' t_x$, for all $t_x \in S'$. If $t_k \leftarrow' t^1$, then t_k serves as t' If $t^1 \leftarrow' t_k$ then t^1 serves as t'. Otherwise, during the mth iteration of algorithm COMPLETE1, an appropriate t' is created such that $t' \leftarrow' t^1$, and $t' \leftarrow' t_k$ and $o \in M(E(t'))$.

Stage 2.

The construction of I and D^2 is accomplished by creating the sets $I_1,...I_n$, where n = |N| and I = $I_1 \cup ... I_n$.(note: If $I_i = \{\ \}$, where i < n, then each of $I_{(i+1)},...I_n$ will be empty). To create the set I_k, we need sets N and I_{k-1} .

Let I_p be the previously created set, and I_c be the set we are currently creating.

Each $t_i \in$ I represents an intersection of some types from N. The types in N are ordered on type numbers. Let 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 ∈ I_p do
   Max(t_i) = i;
   Parents(t_i) = {i};
   Subtypes(t_i) = all t_j ∈ N ∪ Z, such that t_j ←' t_i
While I_p ≠ φ
   Let I_c = { }
   For every t ∈ I_p do
      For each t' ∈ I_c do
```

```
            If Parents(t) ⊆ Parents(t') then
               add t' ←' t to D'
         For i = Max(t) + 1 to n do
            if E(t) ∩ E(t_i) ≠ φ then
               create a new type t_k
               add t_k to I_c
               add t_k ←' t_i and t_k ←' t to D'
               Let Parents(t_k) = Parents(t) ∪ {i}
               For each t_m ∈ ∩ Subtypes(t_j) where t_j ∈ Parents( t_k ) do
               Add t_m ←' t_k to D'
               Let Max(t_k) = i.
      Let I = I ∪ I_c
      Let I_p = I_c
   CLOSE(top);
end COMPLETE2.
```

Consider $S^1 = (N \cup I, B, \leftarrow^1, T, \perp, \phi)$.

Let $\leftarrow^1$ be defined as, $t_i \leftarrow^1 t_j$

if $t_i \leftarrow t_j$, $t_i, t_j \in N$, or $t_i \leftarrow^1 t_j$, $t_i, t_j \in N \cup I$.

Thus, $\leftarrow^1$ is closed on $N \cup I$. And for all t_1 , $t_2 \in N \cup I$, there exists an intersection type t', such that $t' \in N \cup I$.

For S^1 to be a complete type structure the intersection type t' of t_1 and t_2 must represent the true intersection of t_1 and t_2.

1. $\phi(t')$ contains all $o \in \phi(t_1) \cap \phi(t_2)$, because for all o, there exists a type t_o with $o \in \phi(t_o)$ and $t_o \leftarrow^1 t_1$, $t_o \leftarrow^1 t_2$ (by algorithm COMPLETE1)!.

   $t_o \leftarrow^1 t'$, by construction (algorithm COMPLETE2), thus $o \in \phi(t')$.

2. $\phi(t')$ contains nothing but $\phi(t_1) \cap \phi(t_2)$, because each $t_e \leftarrow^1 t'$ is either a user defined type or an extra type. If t_e is a user defined type then the relations $t_e \leftarrow^1 t_1$ and $t_e \leftarrow^1 t_2$ exist. Also $\phi(t_e) \subseteq \phi(t_1) \cap \phi(t_2)$. If t_e is an extra type, then it is created because an object $o \in \phi(t_1) \cap \phi(t_2)$ exists and $o \in \phi(t_e)$.

Thus S is a complete type structure such that

for all $t_1, t_2 \in N'$ there exists $t' \in N'$ such that

$\phi(t') = \phi(t_1) \cap \phi(t_2)$.

Termination :

Algorithms COMPLETE1 and COMPLETE2 both iterate on the same basic loop condition, i.e., until Z_p or I_p is empty. The set Z_p or I_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

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

$S' = (N', B, \leftarrow', T, \perp, \theta')$

S and S' are equivalent with respect to any N^* , where

$N^* \subseteq N \cap N'$, iff

for all $t \in N^*$, for all $o \in B$: $o \in \theta(t)$ iff $o \in \theta'(t)$.

Operations that retain equivalence between type structures:

1. Adding a new type.

2. Adding a link between types $t_i \leftarrow t_j$, iff for all $o \in \theta(t_i)$, $o \in \theta(t_j)$

3. Adding a term T to a type t iff : for all $o \in M(T)$, 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.

7.Complexity.

In this section, we discuss the complexity of the modified unification algorithm and the typed unification mechanism.

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 section 6. 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 depends 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 unifcations 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) = the expected number of new glbs needed to be created to unify the kth pair of literals.

$p_k(i)$ = the probability that the glb for the ith unification of terms in the kth unification of literals does not already exist.

Therefore,

$$E(k) = \sum_{i=1}^{m} p_k(i) * 1$$

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 $E(k) = m(1-r)^k$,

where $r = \frac{m}{N}$.

Base case:

$E(1) = m * 1 = m$, 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 $E(k+1) = m(1-r)^k$

$$E(k+1) = m\frac{(N-(m+m(1-r)\ldots\ldots+m(1-r)^{k-1}))}{N}$$

$$= m(1 - r(1 + (1 - r) + \ldots\ldots(1 - r)^{k-1}))$$

$$= m\frac{(1 - r(1 \times (1 - (1 - r)^k)))}{(1 - (1 - r))}$$

$$= m(1 - 1 + (1 - r)^k)$$

$$= m(1 - r)^k$$

Thus, taking the limit of E(k) as $k \rightarrow \infty$,

$$= \lim_{k \rightarrow \infty} m(1 - r)^{k-1}$$

$$= 0 \text{ as } (1 - r) < 0$$

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 $O(2^n)$.

8. Example.

Here we show an example run of the type inheritance mechanism.

<u>Example.</u>

Consider the following program segment. See Figure 2 for a diagram of 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 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)

9.Discussion and Conclusions.

We have created a Prolog compiler incorporating function evaluation and type inheritance into unification. This was achieved using the constraint satisfaction model of computation.

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 into unification in this Prolog compiler. compiler. We have also showed the soundness and termination properties of our type implementation.

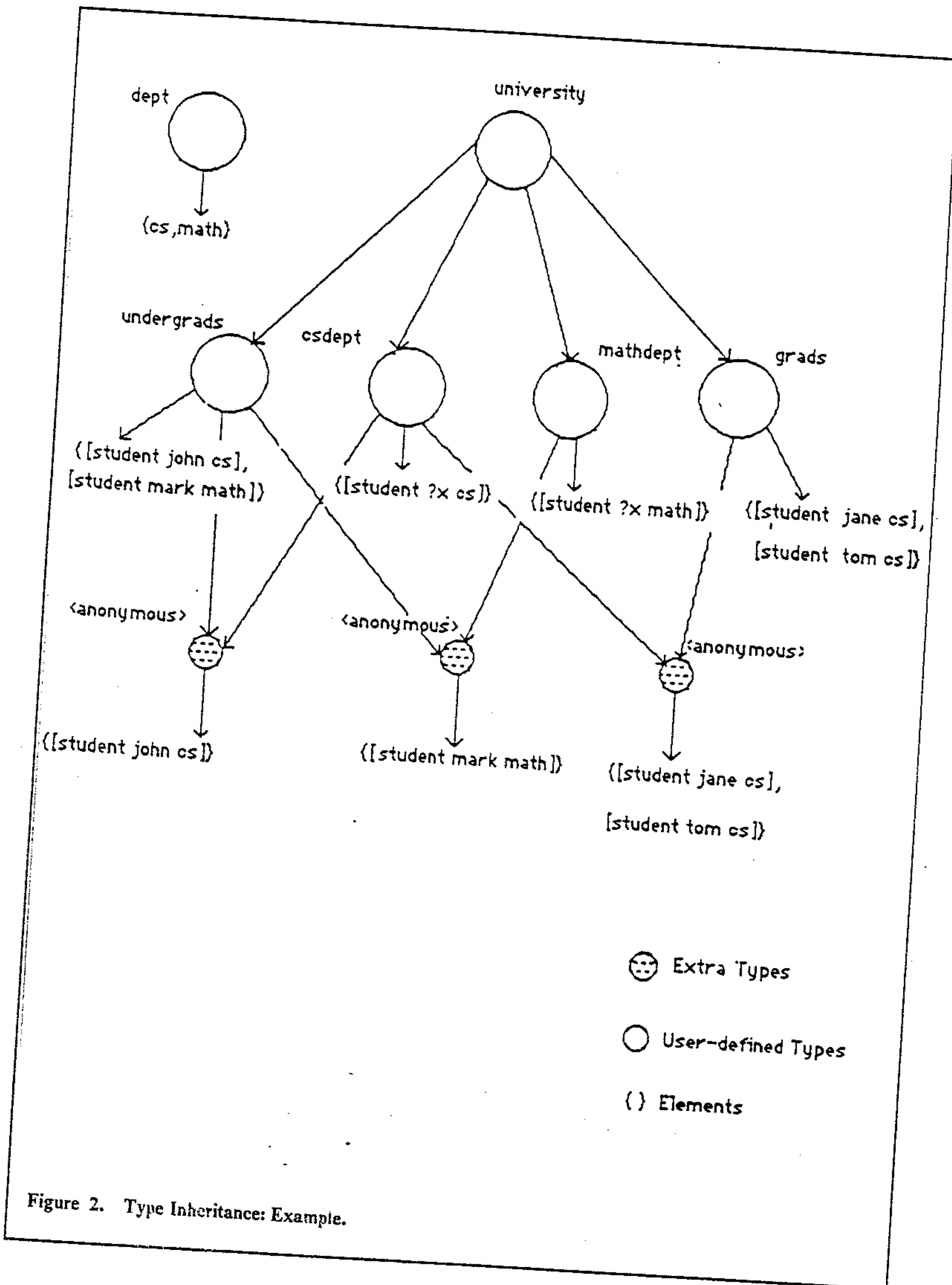


Figure 2. Type Inheritance: Example.

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.

References:

[AITK 86] H.Ait-Kaci and R.Nasr **Login : A Logic Programming Language With Built-In Inheritance** , J. Logic Programming, 1986.

[AITK 88] H.Ait-Kaci,R.Nasr,J.Seo **BABEL : A Base for an Experimental Library**,ACM,1988.

[COLM 84] A.Colmerauer, **Equations and Inequations on Finite and Infinite Trees**, Proceedings of the International Conference on Fifth Generation Computer Systems, 1984.

[COLM 87] A.Colmerauer, **Opening the Prolog III Universe** , BYTE, August 1987.

[DAVI 87] E.Davis, **Constraint Propagation with Interval Labels**, Artificial Intelligence, Vol 32, 1987. 1985.

[HUBE 87]

[JAFF 87] J.Jaffar,J-L.Lassez, **Constraint Logic Programming**, Proceedings of the Conference on the Principles of Programming Languages, Munich 1987.

[LASS 87] C.Lassez **Constraint Logic Programming** ,BYTE, August 1987.

[MACK 77] A. K. Macksworht, "Consistency in networks of relations," **Artificial Intelligence**, 8, 99-118.

[MACK 85] A. K. Macksworth, J. A. Mulder, W. S. Havens, "Hierarchical are consistency: exploiting structured domains in constraint satisfaction problems," **Computational Intelligence**, 1, 118-126.

[ROAC 89] J.Roach, R.Sundararajan and L.T.Watson, **Replacing Unification by Constraint Satisfaction to Improve Logic Programming Expressiveness**, Journal of Automated Reasoning.

[WALT 87] C.Walther, **A many-sorted calculus based on resolution and paramodulation**,Research Notes in AI, Pitman,London, and Morgan Kaufman, Los Angeles, Ca, 1987.

[WALT 88] C.Walther,**Many-Sorted Unification** ,JACM, Vol 35, No 1,January 1988,pp 1-17.

[WARR 83] D.H.D.Warren, **An Abstract Prolog Instruction Set**, TR-309,SRI Project 4776,Menlo Park,CA, Stanford Research Institute 1983.

Appendix A. 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 '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 $(n + 2) * 32$ 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_i, A_n, **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_n, trails A_n if not (S2 < S1), and sets its type to the GLB(S1,S2). Then A_n is stored in X_i.

GetTypedPVar V_i,A_n, **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_n, trails A_n if not (S2 < S1), and sets its type to the GLB(S1,S2). Then A_n is stored in V_i.

GetStructure F, A_i, **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_i 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 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 GLB(S1,S2).

Put Instructions.

PutTypedTVar X_i,A_n, **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_n.

PutTypedPVar V_i,A_n, 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_n.

PutStructure F, A_i, 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 corraesponding structure pointer into register A_i . Execution continues in "write mode".

Unify Instructions.

UniTypedTVar A_i, 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_i 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. A_i is made a reference to S.

UniTypedPVar V_i, 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_i 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_i 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 < < carvivores)

(venusflytrap < < plants)

(pitcherplant < < carvivores)

(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: annonymous),

   {* where annonymous 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

6 - Wild

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.
```

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Field	Entry
1a. REPORT SECURITY CLASSIFICATION	Unclassified
1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
3. DISTRIBUTION/AVAILABILITY OF REPORT	Unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	Systems Research Center SRC-90-003
5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION	Systems Research Center
6b. OFFICE SYMBOL (If applicable)	
6c. ADDRESS (City, State, and ZIP Code)	320 Femoyer Hall, Virginia Tech, Blacksburg, Virginia 24061-0251
7a. NAME OF MONITORING ORGANIZATION	Naval Surface Warfare Center
7b. ADDRESS (City, State, and ZIP Code)	Dahlgren, Virginia 22448-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	Naval Surface Warfare Center
8b. OFFICE SYMBOL (If applicable)	
8c. ADDRESS (City, State, and ZIP Code)	Dahlgren, Virginia 22448-5000
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Hierarchical Constraint Satisfaction as a Model for Adding Types with Inheritance to Prolog

12. PERSONAL AUTHOR(S)
Chandan Chitale, John Deighan, and John Roach

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 2/10/89 TO 2/9/90	May 1989	48

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

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. Prolog operates on the Herbrand universe, a single unstructured domain. In problems with large structured domains, the number of resolution steps required may become large. We have incorporated type inheritance into Prolog to exploit large structured domains to write more concise code and to obtain shorter proofs. Types are subuniverses corresponding to sets of objects. The subset of relation between types induces a hierarchy on the universe. We use hierarchical constraint satisfaction concept to incorporate these extensions into Prolog. We also provide a formal proof that our typed unification algorithm is correct. Our implementation of hierarchically constrained unification extends standard Prolog and directly augments the Warren Abstract Machine (WAM) concept.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified