# Logic and Inheritance

*Hassan Aït-Kaci*
*Roger Nasr*

*Microelectronics and Computer Technology Corporation*
*9430 Research Boulevard*
*Austin, TX 78759*

## 1 Abstract

An elaboration of the Prolog language is described in which the notion of first-order term is replaced by a more general one. This extended form of terms allows the integration of inheritance—an *IS-A* taxonomy—directly into the unification process rather than indirectly through the resolution-based inference mechanism of Prolog. This results in more efficient computations and enhanced language expressiveness. The language thus obtained, called LOGIN, subsumes Prolog, in the sense that conventional Prolog programs are equally well executed by LOGIN.

**Acknowledgements:** We wish to thank Bob Boyer, Matthias Felleisen, and Fernando Pereira for their constructive feedback on the contents of this paper.

## 2 Introduction

Inheritance can be well captured in logic by the semantics of logical implication. Indeed,

$$\forall x.\, Whale(x) \Rightarrow Mammal(x)$$

is *semantically* satisfactory. However, it is not *pragmatically* satisfactory. In a first-order logic deduction system using this implication, inheritance from "mammal" to "whale" is achieved by an *inference* step; whereas, the special kind of information expressed in this formula somehow does not seem to be meant as a deduction step—thus *lengthening* proofs—but rather, as a means to accelerate, or focus, a deduction process—thus *shortening* proofs.

Many proposals have been offered to deal with inheritance and taxonomic information in Automated Deduction. Admittedly, expressing everything in first-order logic as proposed in [6], and [4], is semantically correct. Nevertheless, these approaches dodge the issue of improving the operational deduction process. Other more operational attempts, like those reported in [9], and [5], propose the use of some forms of semantic network. However, it is not always clear what semantics to attribute to these formalisms which, in any case, lose the simple elegance of Prolog-like expressiveness.

As shown in [2], the syntax and operational interpretation of first-order terms can be extended to accommodate for taxonomic ordering relations between constructor symbols. As a result, we propose a simple and efficient paradigm of unification which allows the separation of (multiple) inheritance from the logical inference machinery of Prolog.

We introduce in Section 3 the flavor of what we believe to be a more expressive and efficient way of using taxonomic information, as opposed to straight Prolog. Then, in Section 4, we discuss the adequacy of the use of first-order terms in Prolog. This leads to a proposal in Section 5 to embody taxonomic information as record-like type structures. An efficient unification algorithm which computes greatest lower bounds of such structures is then described in Section 6. Finally, in Sections 7 and 8, a sim-

ple interpreter is illustrated on examples, showing how this extended notion of terms may be integrated into the SLD-resolution mechanism of Prolog.

# 3 Motivation

Let us consider the following example:

> *It is known that all persons like themselves. Also, students are persons, and the individual John is a student.*

This simple information can be expressed in first-order logic as:

$$\begin{array}{ll} & \forall x.\ person(x) \Rightarrow likes(x,x) \\ \& & \forall x.\ student(x) \Rightarrow person(x) \\ \& & student(john). \end{array}$$

and thus in Prolog by:

```
likes(X,X) :- person(X).
person(X) :- student(X).
student(john).
```

To check whether John likes himself is hence:

```
?- likes(john,john).
Yes
```

On the other hand, we can equivalently represent the information above in *typed* first-order logic as follows:

$$\begin{array}{ll} & \forall x \in person.\ likes(x,x) \\ \& & student \subset person \\ \& & john \in student. \end{array}$$

Now, if *type checking* (*i.e.*, that one set is the subset of another, or one element belongs to a set), can be done efficiently, then the typed translation can achieve better performance, with no loss of semantics. Indeed, in our little example, to infer that John likes himself is immediate—one application of *modus ponens* rather than two in this case. This simple idea can be made practical, and is the basis of the extension of Prolog we are presenting in this document.

In Prolog, a resolution step involves a state transition with context saving, variable binding, *etc.*, and is therefore costly. The simple kind of logical implication in the above problem should not contribute to the length of a deduction. Indeed, in this example it should be immediate to see that by virtue of being a student John is also a person. It would be convenient if one could *declare* that, in the unification process, the symbol *student* can match the symbol *person*. Such declarations could look like

```
student < person.
john < student.
```

where the symbol '<' stands for "is a".

In view of these declarations, the original problem could be reformulated using a typing notation such as

```
likes(X : person, X).
```

Then, by a *unification* step rather than by a *resolution* step, the previous answer follows for the query:

```
?- likes(john,john).
```

Thus, we can view the process of unification as computing a greatest lower bound of two symbols relative to the $<$ ordering.

This simple example may not be convincing as a true improvement. That is, one may argue that trading a unification step for a resolution step is not worthwhile. However, unification with inheritance is by far more economical than resolution. Even if this were not the case—*i.e.*, the two steps of computations were equally costly—as the length of inheritance chains increases the motivation for using fast unification rather than resolution appears more clearly.

# 4 An Operational Interpretation of Terms

In the logic interpretation of Prolog, functional first-order terms which are not variables appear as Skolem constants or functions. However, in Prolog, such *functions* are never evaluated. Rather, they are used *operationally* as *type constructors*. The most known example is the famous *cons* list constructor; but a Prolog user can take advantage of this operational interpretation of terms for organizing data, as for instance, in database applications of Prolog [8].

As a result, Prolog's operational use of first-order terms makes them behave as *record structures*; *e.g.*, a term as $person(x,y,z)$ is seen as a three-field record, whose fields may be given some conventional interpretation by the programmer (say, first argument is name, second is date of birth, and third is sex). The implicit operational semantics of such a constructor term is that it denotes the set (type) of all "person" records in the database.

Thus, unification of first-order terms becomes a simple-minded *inheritance* operation, as variables in terms act as slots which are filled as they become instantiated. A subtype of

$$person(x,y,z)$$

a generic denotation of the set of all (records of) persons in the database, may thus be

$$person(name(john,x),y,male)$$

a generic denotation of the set of all (records of) male persons with first name John in the database. In fact, under this interpretation of terms as types, unification is interpreted as *intersection* of types. For example, the intersection of the set of persons whose last name is the

same as their first name

$$person(name(x,x),y,z)$$

with the set of all male persons whose first name is John

$$person(name(john,x),y,male)$$

must be indeed the set of all male persons named John John

$$person(name(john,john),x,male).$$

Since they are not operationally used as functions, Prolog first-order terms suffer from undeserved limitations in their syntax, a legacy of their original functional semantics.

Looking at first-order terms purely syntactically in their use as type constructors, one finds that *fixed arity* of signature symbols is an irrelevant burden. For example, if after extensive use of a three-field record $person(x,y,z)$, a user realizes that a fourth field (say, social security number) is needed, all previous occurrences of the *person* record must be revised and given a fourth argument [6].

Another limitation, which is also a corollary of fixed-arity, is that the interpretation of argument positions is non-transparent to the user. Indeed, in using a *person* record, one must always be aware that the first argument is a name, the second is a date, *etc.* Clearly, the classical explicit labeling of record fields by symbolic keywords is better than implicitly limiting these labels to be ordered ungapped sequences of integers.

The third most fundamental limitation of terms as type structures can be best understood when one ponders upon the respective roles of signature and variable symbols in term unification. A signature symbol is a type constructor and thus acts as an instantiation *filter*. Indeed, unification fails for two non-identical signature symbols. As a result, any further instance of, say, $person(x,y,z)$ must have *person* as root symbol. There is no reason why this filtering role of constructor symbols must be limited to an open/closed behavior. Indeed, $person(x,y,z)$ should be allowed to be further instantiated as $student(x,y,z)$ if the interpretation of the data is such that a "student" type is a subtype of a "person" type. This gradual filtering can be expressed as a partial ordering (type subsumption) on the constructor signature. Hence, unification of signature symbols is now seen as a *greatest lower bound* (GLB) operation. If a signature is augmented with a special *least element* symbol $\bot$ denoting failure of unification, conventional unification of constructor symbols is still a GLB operation.

On the other hand, a variable occurrence means the absence of filter; *i.e.*, it is a *wild card* for term instantiation. As importantly, a variable has a second role as it acts as a *tag* imposing *equality constraints* among subterms—*all* occurrences of the same variable in a given term must be instantiated by identical terms. As an instantiation wild card, a variable behaves as a filter—very permissive, but a filter nonetheless. As a tag, it behaves as an equality constraint. It is a key observation that variables should not carry such a dual information. Firstly, because it is the role of the signature symbols to carry filtering information. And secondly, because even in their equational role, variables are unduly limited. Indeed, as variables are allowed to occur only as leaves in a term, they cannot impose equality constraints *within* the term; *i.e.*, anywhere from root to leaves.

Based on these observations, it comes naturally that the wild card role should be played by a special *greatest element* symbol $\top$ augmenting the signature. As for equality constraints, we propose that variables be called *tags* and allowed to appear anywhere within a term.

All the foregoing limitations are overcome in the syntax of partially-ordered type structures defined next.

# 5 A Calculus of Type Containment

We shall call the syntactic representation of a structured type a *$\psi$-term*. Informally, a $\psi$-term consists of:

1. A *root symbol* which is a type constructor, and denotes a class of objects.
2. *Attribute labels* which are record field symbols, associated with $\psi$-terms. Each label denotes a function *in intenso* from the root type to the type denoted by its associated sub-$\psi$-term. Concatenation of labels denotes function composition.
3. *Coreference* constraints among paths of labels which indicate that the corresponding attribute compositions denote the same functions. In other words, coreference specifies that some functional diagram of attributes must be commutative.

Consider an example of such a $\psi$-term where the root symbol is *person*:

```
person(id => name;
       born => date(day => integer;
                    month => monthname;
                    year => integer);
       father => person)
```

It has three sub-$\psi$-terms under the attribute labels *id*, *born*, and *father*, respectively. We follow the convention of using identifiers starting with a lower-case letter for type symbols and attribute labels. Identifiers starting with an upper-case letter are *tag* symbols and denote coreference among attribute compositions. An example of a $\psi$-term with tags is:

```
person(id => name(first => string;
                  last => X : string);
       father => person(id => name(last => X : string)))
```

The tag symbol $X$ occurs under *id.last* and *father.id.last*, and indicates a coreference constraint; *i.e.*, identical substructures.

To be consistent, a $\psi$-term's syntax cannot be such that different type structures are tagged by the same tag

symbol. For example, if something else than *string* appeared at the address *father.id.last* in the above example, the $\psi$-term would be ill-formed. Hence, in a well-formed $\psi$-term—or *wft*, for short—we shall omit writing more than once the type for any given tag. For instance, the above $\psi$-term will rather be written as:

```
person(id => name(first => string;
                  last => X : string);
       father => person(id => name(last => X))).
```

In particular, this convention allows the concise representation of *infinite* structures such as:

```
person(id => name(first => string;
                  last => string);
       father => X : person(son => person(father => X)))
```

where a cyclic coreference is tagged by $X$.

The type signature $\Sigma$ is a partially-ordered set of symbols. Such a signature always contains two special elements: a greatest element ($\top$) and a least element ($\bot$). Type symbols denote sets of objects, and the partial order on $\Sigma$ denotes set inclusion. Hence, $\top$ denotes the set of all possible objects—the *universe*. We shall omit writing the symbol $\top$ explicitly in a wft; by convention, whenever a type symbol is missing, it is understood to be $\top$. For example, in the wft:

```
person(id => (first => string;
              last => X);
       father => person(id => name(last => X))).
```

$\top$ is the type symbol occurring at addresses *id*, *id.last*, and *father.id.last*.

On the other hand, $\bot$ denotes the empty set and is the type of no object. Consequently, $\bot$ may appear in no wft other than $\bot$, since that would entail that there be no possible object for the corresponding attribute. As a result, any $\psi$-term with at least one occurrence of $\bot$ is identified to $\bot$. Finally, since the information content of tags is simply to impose coreference constraints, it is clear that any one-to-one renaming of a wft's tags does not alter the information content of the wft. In summary, we shall consider wfts up to the above syntactic remarks to be structured type expressions.

The information content of type structures such as those whose syntax is informally introduced above can be formally defined. Namely, a wft structure can be seen as the conjunction of three mathematical abstractions:

1. A $\psi$-term domain $\Delta$—a *regular* set of finite strings of $\mathcal{L}^*$ closed under the *prefix* operation.

2. A coreference relation $\kappa$—an equivalence relation on $\Delta$ of finite index, which is *right-invariant* for label concatenation. That is, the number of coreference classes is finite; and, whenever two addresses corefer, any pair of addresses in the domain obtained from them by further concatenation on the right must also corefer.

3. A type function $\psi$—extending a partial function defined from the set of coreference classes $\Delta/\kappa$ to the type signature $\Sigma$ by associating the type symbol $\top$ to all strings of $\mathcal{L}^*$ which are not in $\Delta$.

Thus, a wft is precisely formalized as such a triple $\langle \Delta, \kappa, \psi \rangle$.

The partial order on the type signature $\Sigma$ is extended to the set of wfts in such a way as to reflect set inclusion interpretation. Informally, a wft $t_1$ is a *subtype* of a wft $t_2$ if:

1. The root symbol of $t_1$ is a subtype in $\Sigma$ of the root symbol of $t_2$;

2. all attribute labels of $t_2$ are also attribute labels of $t_1$, and their wfts in $t_1$ are subtypes of their corresponding wfts in $t_2$; and,

3. All coreference constraints binding in $t_2$ must also be binding in $t_1$.

For example, if $\Sigma$ is such that *student* $<$ *person* and *austin* $<$ *cityname*, then the wft

```
student(id => name(first => string;
                   last => X : string);
        lives_at => Y : address(city => austin);
        father => person(id => name(last => X);
                         lives_at => Y))
```

is a subtype of the wft

```
person(id => name(last => X : string);
       lives_at => address(city => cityname);
       father => person(id => name(last => X))).
```

This partial ordering on wfts is formally defined as follows. A wft $t_1$ is a subtype of a wft $t_2$ if and only if:

*either* $t_1 = \bot$;

*or* $t_1 = \langle \Delta_1, \kappa_1, \psi_1 \rangle$, $t_2 = \langle \Delta_2, \kappa_2, \psi_2 \rangle$, and:

1. $\Delta_2 \subseteq \Delta_1$
2. $\kappa_2 \subseteq \kappa_1$
3. $\forall u \in \mathcal{L}^*,\ \psi_1(u) \leq \psi_2(u)$.

In fact, a stronger result is proved in [2]. Namely, if the signature $\Sigma$ is such that GLBs (respectively, LUBs)exist for all pairs of type symbols with respect to the signature ordering, then GLBs (LUBs) also exist for the extended wft ordering. In other words, the wft ordering extends a (semi-)lattice structure from the signature to the wfts. As an example, if we consider the signature of Figure 1, then the LUB of the wft

```
child(knows => X : person(knows => queen;
                          hates => Y : monarch);
      hates => child(knows => Y;
                     likes => wicked_queen);
      likes => X)
```

and the wft

```
adult(knows => adult(knows => witch);
      hates => person(knows => X : monarch;
                      likes => X))
```

is the wft

```
person(knows => person;
       hates => person(knows => monarch;
                       likes => monarch))
```

and their GLB is the wft

```
teenager(knows => X : adult(knows => wicked_queen;
                            hates => Y : wicked_queen);
         hates => child(knows => Y;
                        likes => Y);
         likes => X).
```

The conventional case of first-order terms is just a particular restriction of the above. Namely, first-order terms are $\psi$-terms such that:

1. the signature is a *flat* lattice—*i.e.*, such that all the symbols, except for $\top$ and $\bot$, are incomparable;
2. tags may appear only at leaf level, and when so, only with the symbol $\top$; and,
3. attribute labels are fixed initial sequences of natural numbers for each signature symbol.

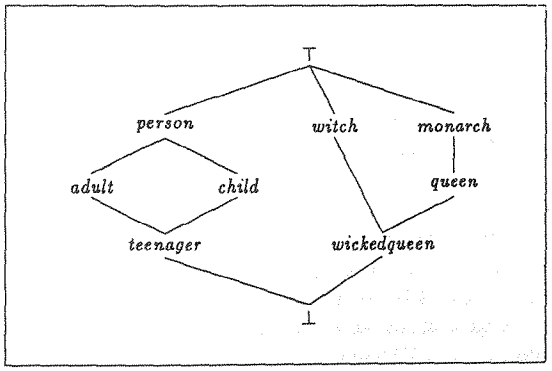


Figure 1: A Signature Which Is a Lattice

Furthermore, the GLB is given by first-order unification [11], and the LUB is given by first-order generalization [10].

For the purpose of integrating logic programming and inheritance, all we shall need is $\psi$-term unification. We shall assume that the signature is a lower semi-lattice—*i.e.*, GLBs exist for all pairs of type symbols. We need an algorithm which, given any pair of wfts, will compute the greatest wft which is a subtype of both wfts. This is explained next.

# 6 The Unification Algorithm

We shall now describe an efficient $\psi$-term unification algorithm. This algorithm computes the $\psi$-term which is the greatest lower bound of two given $\psi$-terms. It assumes that the constructor symbol signature is a lower semi-lattice. The case where the signature is not a lower semi-lattice presents no real problem; for details, the reader is referred to [3].

The $\psi$-term unification algorithm was originally given and proven correct in [2]. It uses the same idea as the method used by Huet [7] for unification of regular first-order terms based on a fast procedure for congruence closure. However, Huet's algorithm is devised for conventional (*i*) fixed-arity terms with (*ii*) arguments identified by position, and (*iii*) over flat signatures. The algorithm presented next does not impose these stringent restrictions.

In order to facilitate the presentation of the algorithm, we first introduce some notation. Although the syntax of $\psi$-terms allows ellipsis of unshared tags—*i.e.*, at those addresses which are alone in their coreference class—it is clear that all addresses in a wft could be explicitly tagged. Let $s$ and $t$ be two $\psi$-terms to unify such that

$$s = X_0 : f(k_1 \Rightarrow X_1 : s_1, \ldots, k_m \Rightarrow X_m : s_m)$$
$$t = Y_0 : g(l_1 \Rightarrow Y_1 : t_1, \ldots, l_n \Rightarrow Y_n : t_n)$$

and such that $s$ and $t$ have their tags renamed apart; *i.e.*,

$$tags(s) \cap tags(t) = \emptyset$$

where

$$tags(s) = \{X_0\} \cup \bigcup_{i=1}^{m} (\{X_i\} \cup tags(s_i))$$
$$tags(t) = \{Y_0\} \cup \bigcup_{i=1}^{n} (\{Y_i\} \cup tags(t_i))$$

are the sets of all the tags occurring in $s$ and $t$, respectively.

Each tag uniquely identifies a $\psi$-term node containing the following information:

- a tag identifier;
- a type constructor;
- a set of next nodes uniquely indexed by attribute labels.

The tag identifier is superfluous in a real implementation where pointers or store addresses can be used. It is used here only for presentation convenience. We shall use a pseudo-record data structure *tagnode* indexed on tag symbols with the appropriate corresponding fields; *e.g.*,

```
tagnode = record
            id          :  tag symbol;
            type        :  constructor symbol;
            subnodes    :  set of pairs ⟨label, tagnode⟩;
            coreference :  tagnode
          end
```

The field *coreference* carries information about coreference class membership. Indeed, since the unification algorithm is a node-merging process which coalesces tag nodes, it is also necessary to represent coreference classes as disjoint sets of tag nodes. Such disjoint sets are represented using an inverted tree representation so that each coreference class may be uniquely identified by one of its node elements—the class representative. (The reader not familiar with the UNION/FIND problem is referred to [1], pp. 129-145.) Hence, a tagnode is its class representative if and only if its *coreference* is nil.

Two operations on tag nodes are defined:

- $FIND(x)$ returns the class representative of tag node $x$;
- $UNION(x, y, z)$ performs the set union of the two—disjoint—classes represented by $x$ and $y$, and whose result has representative $z$.

Also, we define

$$labels(x) = \{l \mid \exists y \ \langle l, y\rangle \in x.subnodes\}$$

to denote the set of attribute labels attached to the tag node $x$.

Given a tag node $x$ and an attribute label $l$ in $labels(x)$, $subterm(x, l)$ denotes the tag node under attribute $l$ of $x$; *i.e.*,

$$\langle l, y\rangle \in x.subnodes \Rightarrow subterm(x, l) = y$$

In the algorithm of Figure 2, a tag node *id* stands for the tag node itself. Initially, since no merging has yet occurred, all tag nodes in $tags(s) \cup tags(t)$ have *coreference* set to nil (*i.e.*, each tag node is alone in its class).

Informally, this unification algorithm follows through all possible attribute paths in both $\psi$-terms. Pairs of tag nodes that are reached following the same path of attributes labels are merged into coreference classes. Each class has a unique representative (given by *FIND*) where all information relative to the class is gathered. In particular, type symbols are coerced by the GLB operation ($\wedge$) on the signature $\Sigma$; and attribute labels of a node being merged must be carried to the representative of the class. This latter procedure is described in Figure 3.

The unification procedure returns either $\perp$ if a clash of type constructor occurs; or, the $\psi$-term built out of the merged graph of tag nodes. This is what the *REBUILD* procedure does, as explicated in Figure 4. Each merged

```
procedure UNIFY(s, t);
  begin
  PAIRS ← {⟨X0, Y0⟩};
  while PAIRS ≠ ∅ do
    begin
    remove ⟨x, y⟩ from PAIRS;
    u ← FIND(x);
    v ← FIND(y);
    if u ≠ v then
      begin
      σ ← u.type ∧ v.type;
      if σ = ⊥ then return(⊥)
      else
        begin
        UNION(u, v, w);
        w.type ← σ;
        for each l in labels(u) ∪ labels(v) do
          begin
          if w = v
            then CARRYLABEL(l, u, v)
            else CARRYLABEL(l, v, u);
          if l ∈ labels(u) ∩ labels(v) then
          PAIRS ← PAIRS ∪ {⟨subterm(u, l), subterm(v, l)⟩}
          end
        end
      end
  return(REBUILD(tags(s) ∪ tags(t)))
  end
```

Figure 2: The $\psi$-term Unification Procedure

```
procedure CARRYLABEL(l, u, v);
  begin
  if l ∉ labels(v)
    then v.subnodes ← v.subnodes ∪ {⟨l, FIND(subterm(u, l))⟩}
  end
```

Figure 3: The *CARRYLABEL* Instruction

class is attributed a new tag node carrying the information assembled by the unification procedure at the class representative nodes.

The algorithm of Figure 2 is a variation on the algorithm deciding equivalence of two finite-state automata (see [1]). It computes the least coreference relation on attribute label strings which is *right-invariant* for concatenation of labels, and contains both coreference relations of the given $\psi$-terms (see [2]).

Provided that two simple rules of computation for the *UNION* and *FIND* operations are observed which keep inverted trees as *balanced* and as *shallow* as possible (see [1]), the algorithm of Figure 2 has a time complexity of order *almost* linear in $n$, the total number of nodes (*i.e.*, $n = |\, tags(s) \cup tags(t)\,|$). In fact, it has a worst case upper bound of $O(nG(n))$, where $G$ grows *very* slowly—of the order of an inverse of the Ackerman function. In particular, $G(n) \leq 5$ for all practical purposes!

The reader is encouraged to trace the algorithm on an example. A detailed trace is described in [3]. Next, we illustrate LOGIN on two examples.

```
procedure REBUILD(tagset);
  begin
  CLASSES ← ∪_{x∈tagset}{FIND(x)};
  for each x in CLASSES do ID[x] ← NewTagSymbol;
  for each x in CLASSES do
    begin
    NODE ← NewTagNode;
    with NODE do
      begin
      id ← ID[x];
      type ← x.type;
      subnodes ← {⟨l, ID[FIND(y)]⟩ | ⟨l, y⟩ ∈ x.subnodes};
      coreference ← nil
      end
    end
  return(ID[FIND(X_0)])
  end
```

Figure 4: The *REBUILD* Procedure

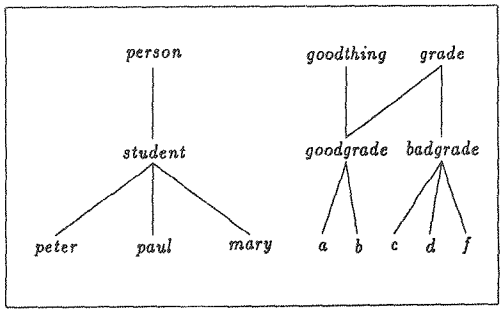


Figure 5: A Signature for the Simple Example

# 7 A Simple Example

LOGIN is simply Prolog where first-order terms are replaced by $\psi$-terms. Thus, we shall simply show that the skeleton of a Prolog interpreter implementing a top-down/left-right backtracking search procedure can be adapted in a straightforfard manner. The unification procedure is simply replaced by $\psi$-term unification, altered to allow for undoing coreference merging and type coercion upon backtracking.

Let us consider the following simple example. The signature of Figure 5 is declared in LOGIN by:

```
student < person.
{peter, paul, mary} < student.
{goodgrade ,badgrade} < grade.
goodgrade < goodthing.
{a, b} < goodgrade.
{c, d, f} < badgrade.
```

This essentially expresses the facts that a student is a person. Peter, Paul, and Mary are students. Good grades and bad grades are grades. A good grade is also a good thing. 'A' and 'B' are good grades; but, 'C', 'D', and 'F' are bad grades.

In this context, we can define the following facts and rules.

It is known that all persons like themselves. Also, Peter likes Mary; and, all persons like all good things. Thus, in LOGIN,

```
likes(X : person, X).
likes(peter,mary).
likes(person,goodthing).
```

Peter got a 'C'; Paul got an 'F', and Mary an 'A'. Thus,

```
got(peter,c).
got(paul,f).
got(mary,a).
```

Lastly, it is known that a person is happy if she got something which she likes. Alternatively, a person is happy if he likes something which got a good thing. Hence,

```
happy(X : person) :- likes(X,Y),got(X,Y).
happy(X : person) :- likes(X,Y),got(Y,goodthing).
```

Mary is happy because she likes good things, and she got an 'A'—which is a good thing. She is also happy because she likes herself, and she got a good thing. Peter is happy because he likes Mary, who got a good thing. Thus, a query asking for some "happy" object in the database will yield:

```
?- happy(X).
X = mary
```

The way LOGIN finds the first answer ($X = mary$) is as follows. The invoking resolvent $happy(X : \top)$ unifies with the head of the first defining rule of the *happy* procedure, by coercing $X : \top$ to $X : person$. The new resolvent thus is:

$$likes(X : person, Y), got(X, Y). \qquad (1)$$

Next, $likes(X : person, Y)$ unifies with the first alternative of the definition of *likes*, confirming the previous type coercion of $X : person$, and merging coreference $Y : \top$ to $X : person$. The resolvent thus obtained is:

$$got(X : person, X).$$

This is not matched by anything in the database; and so, LOGIN must backtrack, reinstating the previous coercions and coreferences of resolvent (1).

As a next choice, $likes(X : person, Y)$ unifies with the second alternative of the definition of *likes*, further coercing $X : person$ to $X : peter$, and coercing $Y : \top$ to $Y : mary$. This produces the new resolvent:

$$got(X : peter, Y : mary).$$

This literal finds no match in the database; and so, LOGIN must backtrack again, reinstating the previous coercions of resolvent (1).

The third possible match is the last definition for the predicate *likes*, whereby $Y : \top$ is coerced to $Y : goodthing$. This yields the resolvent:

$$got(X : person, Y : goodthing).$$

For this, the only successful match is the third definition of the *got* predicate. This yields the empty resolvent, and the final result $X = mary$.

At this point, if forced to backtrack, LOGIN attempts the next alternative match for the initial invoking resolvent $happy(X : \top)$; namely, the second rule of the *happy* procedure. The next resolvent is thus:

$$likes(X : person, Y), got(Y, goodthing). \qquad (2)$$

A match with the first alternative of the *likes* definition merges $X$ and $Y$. This gives the resolvent:

$$got(X : person, goodthing).$$

And this matches $got(mary, a)$, producing the second result $X = mary$.

If backtracking is forced once again, resolvent (2) is restored. This time, as seen before, establishing the first literal of this resolvent eventually leads to coercing $X : person$ to $X : peter$, and $Y : \top$ to $Y : mary$, resulting in the resolvent:

$$got(Y : mary, goodthing).$$

And this succeeds by matching $got(mary, a)$.

Hence, this third alternative branch of computation succeeds with the final result $X = peter$. The reader is left to convince herself that there is no other solution for that particular query.

The next section illustrates a more complex example involving the presence of attributes.

# 8 A More Complex Example

The example of Section 7 was simple in the sense that it did not illustrate the use of inheritance among complex $\psi$-term objects—*e.g.*, records with attributes. One such example is next described.

In a type signature, such a type symbol as *person* has virtually any possible attribute typed as $\top$. However, it may be desirable to constrain all possible database instances of *person* to be such that particular attribute types be more specific than $\top$.

For example, let us suppose that we want to impose that every legal record instances of *person* in the database must have:

- a field *id* which must be a *name*;
- a field *dob* which must be a *date*; and,
- a field *ss#* which must have:
  - a field *first* which must be string of characters between '000' and '999';
  - a field *middle* which must be string of characters between '00' and '99'; and,
  - a field *last* which must be string of characters between '0000' and '9999'.

We can write this in LOGIN as:

```
person = (id => name;
          dob => date;
          ss# => (first => ['000'...'999'];
                  middle => ['00'...'99'];
                  last => ['0000'...'9999'])).
```

where, *name* is specified as, say:

```
name = (first => string;
        middle => string;
        last => string).
```

and *date* as:

```
date = (day => [1...31];
        month => [1...12];
        year => [1900...2000]).
```

The "$[\alpha \ldots \beta]$" notation is used to denote interval ranges over linearly ordered built-in types. For example, any string of character '*xy...z*' is an instance of the built-in type *string*, ordered lexicographically using, say, ASCII character codes. Thus, any interval of strings is a subtype of *string*; and unification corresponds to intersection. The same applies to types like *integer*, *real*, *etc.*

Now, let us suppose that we also want to specify that a *student* be a subtype of *person*—*i.e.*, that it inherits whatever attribute restrictions imposed on *person*—and that *student* further imposes restrictions on some attributes; *e.g.*, a *student* has a *major* which must be a *course*, and further constrains the *dob* field to be have a *year* between 1950 and 1970. This is achieved by:

```
student = person(major => course;
                 dob => (year => [1950...1970]).
```

Clearly, it must be checked that these type specifications are not inconsistent. And this can be done statically, before running a LOGIN program.

Similarly, we could elaborate the rest of the signature. For example,

```
employee = person(position => jobtitle;
                  salary => integer).

workstudy < employee.
workstudy < student.
```

```
s1 = student(id => (first => 'John';
                    last => 'Doe');
             major => computerScience;
             ss# => (first => '897';
                     middle => '23';
                     last => '5876')).

w1 = workstudy(id => (first => 'Abebe';
                      middle => 'Nmougoudou';
                      last => 'Bekila');
               major => physicalEducation;
               ss# => (first => '999');
               salary => 10000).
```

Note that inheritance allows for ellipsis of information in the particular records of individuals in the database like $s1$ and $w1$.

Now, we can define facts and rules in the context of this signature. For instance, part of a course enrollment relation could be:

```
takes(s1,[cs101,cs121,ma217]).
takes(w1,[pe999]).
```

To express that all students taking less then 3 courses are considered part-time students, we write:

```
parttime(X : student) :-
    takes(X,CL),length(CL,L), L<=3.
```

where length is trivially defined and computes the length of a list.

Finally, to formulate that all persons whose social security number starts with 999 is foreign, we write:

```
foreign(person(ss# => (first => '999'))).
```

Thus, a query asking for the last name of some foreign employee who is also a part-time student, and earns a salary less than 20000, is:

```
query(X : string) :-
    foreign(Y : employee(salary => Z)),
    parttime(Y : student(id => (last => X))),
    Z < 20000.
```

A remark worth making here is that extensive information can be processed statically for LOGIN before run-time. Thus, besides the inheritance type-checking already mentioned, some compile-time coercion must be performed to maintain consistent typing in clauses. Indeed, typing in the above query as we did would result in the automatic coercion of the $\psi$-term under the tag $Y$, transforming it internally into:

```
query(X : string) :-
    foreign(Y : workstudy(salary => Z;
                          id => (last => X)),
    parttime(Y),
    Z < 20000.
```

Thus, should $\perp$ occur in a clause by static type coercion, the clause would be eliminated.

We leave it as an exercise to the reader to verify that an answer to this query for the foregoing data is:

```
?- query(X).
X = 'Bekila'
```

# 9 Concluding Remarks

We have given a summarized description of a semantically sound and operationally practical typed extension of Prolog, where type inheritance *à la* semantic network is cleanly and efficiently realized through a generalized unification process. We have illustrated how this can be achieved on some detailed examples. The language thus obtained is called LOGIN, an acronymic combination of "logic" and "inheritance".

The gain that we feel LOGIN provides over the conventional Prolog language is twofold:

1. the efficient use of taxonomic information, as well as complex objects;

2. a natural way of dealing efficiently with "set at a time" mode of computation essential to database applications.

In addition, we feel that the inheritance model behind LOGIN offers great potential for compile-time consistency checking, and object-oriented computation in a logic programming paradigm. For example, it is possible, at compile-time, to narrow drastically the range of indexing over a large database of individual records to only an appropriate view, based on the types of arguments of the predicates involved in the rules of a querying program.

# Bibliography


[1] Aho, V.A., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] Aït-Kaci, H. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures.* Ph.D. Thesis, Computer and Information Science, University of Pennsylvania. Philadelphia, 1984.

[3] Aït-Kaci, H., and Nasr, R. *LOGIN: A Logic Programming Language With Built-In Inheritance.* Technical Report MCC-AI-068-85, Microelectronics and Computer Technology Corporation, Austin, 1985. *To appear in The Journal of Logic Programming.*

[4] Allen, J.F., and Frish, A.M. "What's in a Semantic Network", In *Proceedings of the 20th Annual ACL Meeting.* Association for Computational Linguistics, 1982.

[5] Brachman, R.J., Fikes, R.E., and Levesque, H.J. "KRYPTON: A Functional Approach to Knowledge Representation", FLAIR Technical Report *N° 16*, Fairchild Lab. for Artificial Intelligence Research, Fairchild Research Center. Palo Alto, May 1983.

[6] Deliyanni, A., and Kowalski, R.A. "Logic and Semantic Networks", *Comm. of the ACM*, 22(3):184-92. 1979.

[7] Huet, G. *Résolution d'Equations dans des Langages d'Ordre 1, 2, ..., ω*. Thèse de Doctorat d'Etat, Université de Paris VII, France. 1976.

[8] Kowalski, R.A. "Logic for Data Description", In *Logic and Data Bases*, Gallaire, H., and Minker, J. (Eds.), pp.77-103. Plenum Press, 1978.

[9] McSkimin, J.R., and Minker, J. "A Predicate Calculus Based Semantic Network for Question-Answering Systems", In *Associative Networks—The Representation and Use of Knowledge by Computers*, Findler, N. (Ed.). Academic Press, New York, 1979.

[10] Reynolds, J.C. "Transformational Systems and the Algebraic Structure of Atomic Formulas", In *Machine Intelligence 5*, Michie, D. (Ed.). Edinburgh University Press, 1970.

[11] Robinson, J.A. "A Machine-Oriented Logic Based on the Resolution Principle", *Journal of the ACM* 12(1):23-41. 1965.