LOGIN: A LOGIC PROGRAMMING LANGUAGE WITH BUILT-IN INHERITANCE

HASSAN AÏT-KACI AND ROGER NASR

▷ 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. ◁

...it is clear that the internal functioning of intelligent assimilation...is constantly reinforced by the causal situations in which the anticipations are followed by effective controls (the success or failure of the swinging depending on whether the hanging object was really a mobile or not, *etc.*).

This does not mean that all the immediate inferences of the preoperational levels have a causal content. They can serve as classifiers.

JEAN PIAGET, *Understanding Causality*

1. INTRODUCTION

Since the early days of research in automated deduction, inheritance has been proposed as a means to capture a special kind of information, viz., taxonomic information. For example, when it is asserted that "whales are mammals", we understand that whatever properties mammals possess should also hold for whales.

Address correspondence to Hassan Aït-Kaci or Roger Nasr, Artificial Intelligence Program, Microelectronics and Computer Technology Corporation, 9430 Research Boulevard, Austin, TX 78759.

THE JOURNAL OF LOGIC PROGRAMMING

Naturally, this meaning of inheritance can be well captured in logic by the semantics of logical implication. Indeed,

$$\forall x.\mathit{Whale}(x) \Rightarrow \mathit{Mammal}(x)$$

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. But the special kind of information expressed in this formula somehow does not seem to be meant as a deduction step—thus *lengthening* proofs. Rather, its purpose seems to be 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, doing it all in first-order logic, as proposed in [5] and [3], is semantically correct; nevertheless, these approaches dodge the issue of improving the operational deduction process. Other, more operational attempts, like those reported in [8] and [4], 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 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.

By means of examples, we introduce in Section 2 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 3, we give a quick formal summary of how first-order terms may be extended to embody taxonomic information as recordlike type structures, together with an efficient type unification algorithm. This leads to a technical proposal for integrating this notion of terms into the SLD-resolution mechanism of PROLOG. We call the resulting language LOGIN. Together with examples, we describe a LOGIN interpreter in Section 4. An appendix is attached in which a formal semantics for LOGIN is sketched, as well as a lattice-theoretic argument showing how LOGIN's inheritance by unification and its "type as set" semantics naturally provide a built-in set abstraction which makes "set at a time" computations possible.

2. MOTIVATIONAL EXAMPLES

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

$$\begin{aligned} & \forall x.\mathit{person}(x) \Rightarrow \mathit{likes}(x, x) \\ \& \quad & \forall x.\mathit{student}(x) \Rightarrow \mathit{person}(x) \\ \& \quad & \mathit{student}(\mathit{john}). \end{aligned}$$

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 ther 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 perfornace, 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 paper.

Let us consider another example:

A student is a person; $p_1, \ldots, p_n$ are persons; $s_1, \ldots, s_m$ are students; p_i and s_j have the property *prop* for some i, $1 \leq i \leq n$, and some j, $1 \leq j \leq m$.

These simple data can be represented in a straightforward way in PROLOG as follows:

```
person(X) :- student(X).
person(p1).
 ...
person(pn).
student(s1).
 ...
student(sm).
prop(pi).
prop(sj).
```

Thus, a query asking for a person having the property *prop* is formulated by

```
?- person(X),prop(X).
```

PROLOG's SLD-resolution engine finds the solutions to this query (namely, $X = p_i$, $X = s_j$), by direct match (p_i) and by one resolution step (s_j) using the rule translating the information that a student is also a person and then matching on the student s_j who has property *prop*.

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

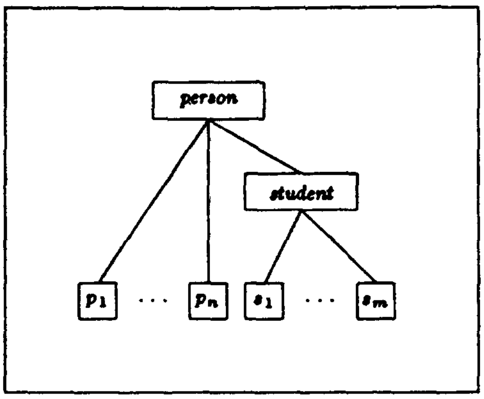

FIGURE 1. A signature for inheritance.

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 s_j 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.
{p1,...,pn} < person.
{s1,...,sm} < student.
```

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

In view of these declarations, the original problem could be reformulated as

```
prop(pi).
prop(sj).
```

We can write the query using a typing notation such as

```
?- prop(X : person).
```

Unification becomes the process of computing the *greatest lower bound* of two symbols relative to the < ordering. Then, by a *unification* step rather than by a *resolution* step, the two previous answers follow.

A declaration of inheritance information such as the above can be graphically visualized as depicted in Figure 1.

We call such a declaration the specification of a *signature*. The signature information is to be used by the unification process in order to realize inheritance. The partial ordering of type symbols such as *person* or *student* in the signature, denotes class containment.

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, as we shall show later, 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 with built-in inheritance appears more clearly. Consider, for example, the following generic problem.

A t_1 is a t_2; a t_2 is a t_3; ... ; a t_{n-1} is a t_n; some individual t is a t_1, and has the property *prop*.

This translates into PROLOG as

```
tn(X) :- tn-1(X).
tn-1(X) :- tn-2(X).
...
t2(X) :- t1(X).
t1(t).
prop(t).
```

Now, asking for a t_n with property *prop* will require n resolution steps before matching on t.

In our new notation, the inheritance information for this generic example can be expressed by the ordering declarations

```
t < t1.
t1 < t2.
...
tn-1 < tn.
```

together with the fact

```
prop(t).
```

Hence, the query

```
?- prop(X : tn).
```

succeeds in *one* step of unification, rather than n resolution steps.

3. EXTENDED FIRST-ORDER TERMS

3.1. An Operational Interpretation of Terms

In first-order logic, a *literal* is a syntactic being of the form:

$$p(t_1, \ldots, t_n)$$

where p is a predicate symbol, and the t_i's are (functional) first-order terms. First-order terms appear ubiquitously in logic and universal algebra. Given a family $\{\Sigma_n \mid n \in \mathbb{N}\}$ of disjoint *ranked signatures* of function symbols, the (functional) signature $\Sigma = \bigcup_{n \in \mathbb{N}} \Sigma_n$ contains symbols to be interpreted as functions in some semantic model, and the subscript n stands for the *arity* (or number of arguments) of the functions these symbols are to denote. Symbols of arity 0 are to denote constants.[1]

Let $\mathscr{V}$ be a nonempty, countably infinite set of variables. Then the set $T_{\Sigma,\mathscr{V}}$ of first-order terms on the signature Σ and the variables $\mathscr{V}$ is defined inductively as

[1] This explains the small technicality of always requiring that Σ_0 be nonempty—which guarantees that potential models are nonempty.

follows:

elements of $\mathscr{V}$ and Σ_0 are first-order terms;

if $f \in \Sigma_n$ and $t_1, \ldots, t_n$ are first-order terms, then $f(t_1, \ldots, t_n)$ is also a first-order term.

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 best-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 [7].

As a result, PROLOG's operational use of first-order terms makes them behave as *record structures*; e.g., a term such 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.

Unification of first-order terms is 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 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 [5].

Another limitation, which is also a consequence of fixed arity, is that the interpretation of argument positions is nontransparent 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 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 nonidentical 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, $person(x, y, z)$ record 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 $\perp$ 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 any filter; i.e., it is a *wild card* for term instantiation. As importantly, a variable has a second role in that 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. 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 natural 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.

3.2. *A Calculus of Partially Ordered Type Structures*

We shall call the syntactic representation of a structured type a *ψ-term*. Informally, a ψ-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 ψ-terms. Each label denotes a function *in intenso* from the root type to the type denoted by its associated sub-ψ-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.

An example of a ψ-term is

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

The root symbol is *person*; it has three sub-ψ-terms under the attribute labels *id*, *born*, and *father*, respectively. We follow the convention of using identifiers starting with a lowercase letter for type symbols and attribute labels. Identifiers starting with an uppercase letter are *tag* symbols and denote coreference among attribute compositions.

An example of a ψ-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.

We shall denote by Σ the set of type constructors (the type *signature*), by $\mathscr{L}$ the set of attribute labels, and by $\mathscr{T}$ the set of tag symbols. Strings of attribute labels are called (ψ-term) *addresses*. Thus, $\mathscr{L}^*$ is the set of all possible addresses. A ψ-term *domain* is the subset of $\mathscr{L}^*$ of the addresses of the ψ-term; e.g., the domain of the ψ-term of the first of the two ψ-terms above is

$\{\varepsilon;$ *id*, *born*, *born.day*, *born.month*, *born.year*, *father* $\}$

where ε denotes the empty string—the root address.

To be consistent, a ψ-term's syntax cannot be such that different type structures are tagged by the same tag symbol. For example, if something other than *string* appeared at the address *father.id.last* in the ψ-term above, it would be ill formed. Hence, in a well-formed ψ-term—or *wft*, for short—we shall omit writing more than once the type for any given tag. For instance, the second of the two ψ-terms above will rather be written:

```
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, as shown in

```
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 Σ is a partially ordered set of symbols. Such a signature always contains two special elements: a greatest element ($\top$) and a least element ($\perp$). Type symbols denote sets of objects, and the partial order on Σ 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, $\perp$ denotes the empty set and is the type of no object. Consequently, $\perp$ may appear in no wft other than $\perp$, since that would entail that there was no possible object for the corresponding attribute. As a result, any ψ-term with at least one occurrence of $\perp$ is identified with $\perp$.

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.

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 ψ-term domain Δ—a *regular* set of finite strings of $\mathscr{L}^*$ closed under the *prefix* operation.

(2) A coreference relation κ—an equivalence relation on Δ 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 ψ—extending a partial function defined from the set of coreference classes Δ/κ to the type signature Σ by associating the type symbol $\top$ to all strings of $\mathscr{L}^*$ which are not in Δ.

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

The partial order on the type signature Σ is extended to the set of wfts in such a way as to reflect the 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 Σ 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 Σ is such that the following ordering holds:

$$\textit{student} < \textit{person}, \qquad \textit{austin} < \textit{cityname}$$

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) $\forall u \in \mathscr{L}^*$, $\psi_1(u) \leq \psi_2(u)$.

In fact, a stronger result is proved in [2]. Namely, if the signature Σ is such that GLBs (respectively, LUBs)[2] 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 2, 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))
```

[2]The acronyms "GLB" and "LUB" stand for "greatest lower bound" and "least upper bound", respectively.

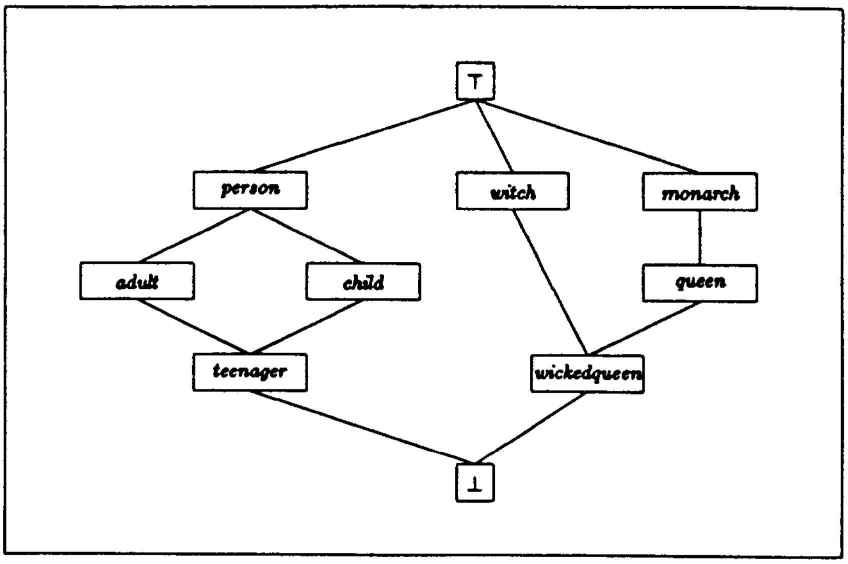

FIGURE 2. A signature which is a lattice.

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).
```

It is not difficult to see that the conventional case of first-order terms is just a particular restriction of the above. Namely, first-order terms are ψ-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.

Furthermore, the GLB is given by first-order unification [10], and the LUB is given by first-order generalization [9].

Thus, in the language that we are about to describe, a first-order term such as

$$f(t_1,\dots,t_n)$$

is nothing but syntactic sugar for the ψ-term

$$f(1 \Rightarrow t_1;\dots;n \Rightarrow t_n).$$

For the purpose of integrating logic programming and inheritance, all we shall need is ψ-term unification. We shall assume that the signature is a lower semilattice —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.

3.3. The Type Unification Algorithm

We shall now describe an efficient ψ-term unification algorithm. This algorithm computes the ψ-term which is the greatest lower bound of two given ψ-terms. It assumes that the constructor symbol signature is a lower semilattice. The case where the signature is not a lower semilattice presents no real problem. Appendix B describes a simple lattice-theoretic construction solving this small technicality.

The ψ-term unification algorithm was originally given and proven correct in [2]. It uses the same idea as the method used by Huet [6] for unification of regular first-order terms based on a fast procedure for congruence closure. However, Huet's algorithm is devised for conventional (1) fixed-arity terms with (2) arguments identified by position, and (3) 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 ψ-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 ψ-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.,

$$\mathit{tags}(s) \cap \mathit{tags}(t) = \varnothing$$

where

$$\mathit{tags}(s) = \{X_0\} \cup \bigcup_{i=1}^{m} (\{X_i\} \cup \mathit{tags}(s_i)),$$

$$\mathit{tags}(t) = \{Y_0\} \cup \bigcup_{i=1}^{n} (\{Y_i\} \cup \mathit{tags}(t_i))$$

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

Each tag uniquely identifies a ψ-term node containing the following information:

a tag identifier;[3]

a type constructor;

a set of next nodes uniquely indexed by attribute labels.

For this reason, we shall use a pseudo-record data structure *tagnode* indexed on tag

[3]This is superfluous in a real implementation where pointers or store addresses can be used. It is used here for presentation convenience and will disappear later.

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[4] so that each coreference class may be uniquely identified by one of its node elements—the class representative. 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.

Giv͏en 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.,

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

In the algorithm of Figure 3, 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 ψ-terms. Pairs of tag nodes that are reached following the same path of attribute 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 Σ; and attribute labels of a node being merged must be carried to the representative of the class. This latter procedure is described in Figure 4.

The unification procedure returns either $\perp$, if a clash of type constructor occurs; or the ψ-term built out of the merged graph of tag nodes. This is what the *REBUILD* procedure does, as explicated in Figure 5. Each merged class is attributed a new tag node carrying the information assembled by the unification procedure at the class representative nodes.

The algorithm of Figure 3 is a variation on the algorithm deciding equivalence of two finite-state automata (see [1]). It computes the least coreference relation on

[4]The reader not familiar with the UNION/FIND problem is referred to [1, pp. 129–145].

```
procedure UNIFY(s, t);
  begin
  PAIRS ← {⟨$X_0$, $Y_0$⟩};
  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
    end
  return(REBUILD(tags(s) ∪ tags(t)))
  end
```

FIGURE 3. The ψ-term unification procedure.

attribute label strings which is *right-invariant* for concatenation of labels, and contains both coreference relations of the given ψ-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 3 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.

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

FIGURE 4. The *CARRYLABEL* instruction.

```
procedure $REBUILD(tagset)$;
  begin
  $CLASSES \leftarrow \bigcup_{x \in tagset}\{FIND(x)\}$;
  for each $x$ in $CLASSES$ do $ID[x] \leftarrow NewTagSymbol$;
  for each $x$ in $CLASSES$ do
    begin
    $NODE \leftarrow NewTagNode$;
    with $NODE$ do
      begin
      $id \leftarrow ID[x]$;
      $type \leftarrow x.type$;
      $subnodes \leftarrow \{\langle l, ID[FIND(y)]\rangle \mid \langle l, y\rangle \in x.subnodes\}$;
      $coreference \leftarrow$ nil
      end
    end
  return($ID[FIND(X_0)]$)
  end
```

FIGURE 5. The *REBUILD* procedure.

Let us illustrate the ψ-term unification algorithm of Figure 3 on a concrete example. Let us take the ψ-term s:

$$\begin{array}{lll} X_0: student(advisor & \Rightarrow X_1: faculty & (secretary \Rightarrow X^3: staff; \\ & & assistant \Rightarrow X_0); \\ roommate & \Rightarrow X_2: employee & (representative \Rightarrow X_3)) \end{array}$$

and the ψ-term t:

$$\begin{array}{lll} Y_0: employee(advisor & \Rightarrow Y_1: f_1 & (secretary \Rightarrow Y_4: employee; \\ & & assistant \Rightarrow Y_5: person); \\ roommate & \Rightarrow Y_2: student & (representative \Rightarrow Y_2); \\ helper & \Rightarrow Y_3: w_1 & (spouse \Rightarrow Y_5)) \end{array}$$

to be the input arguments of *UNIFY*, in the context of the signature shown in Figure 6.[5] Then, the resulting ψ-term yielded by *UNIFY* is

$$\begin{array}{lll} Z_0: workstudy(advisor & \Rightarrow Z_1: f_1 & (secretary \Rightarrow Z_2; \\ & & assistant \Rightarrow Z_0); \\ roommate & \Rightarrow Z_2: workstudy & (representative \Rightarrow Z_2); \\ helper & \Rightarrow Z_3: w_1 & (spouse \Rightarrow Z_0)) \end{array}$$

In order to help the reader follow the effect of this unification procedure on these particular s and t, a description of a trace is detailed in Figure 7, Figure 8, and Figure 9. For simplicity, in this trace we did not observe the two optimization rules for *UNION* and *FIND* alluded to above. We took as a convention always to perform *UNION*(x, y, z) in such a way that $z = y$, i.e., by dereferencing x toward y. Also, it is assumed that the *FIND* operation in this trace has no side effect on the inverted-tree data structures.

[5]By convention, $\top$ and $\perp$ will be implicitly present and omitted in all signatures.

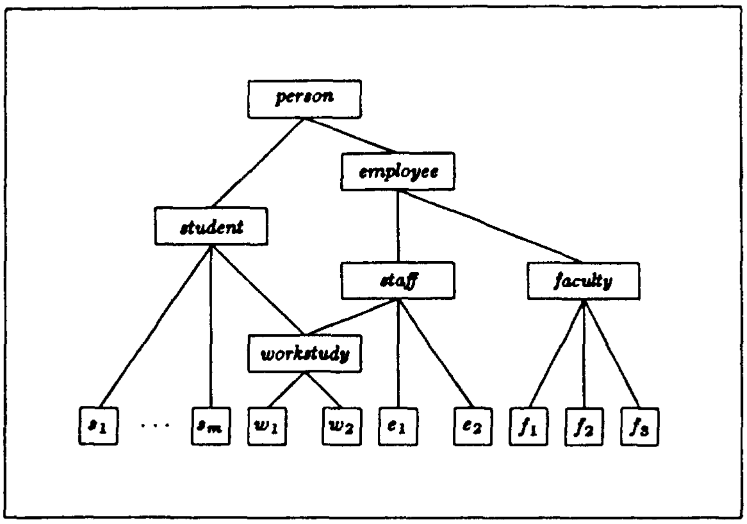

FIGURE 6. A signature with multiple inheritance.

Figure 7 shows the steps of iteration—6 in this case. The second column shows the evolution of the set *PAIRS*.

Figure 8 shows the ultimate dereferencing (via the *coreference* link) at the end of the interactions of *UNIFY*, right before constructing the resulting ψ-term with *REBUILD*.

Figure 9 shows the effect of the *REBUILD* procedure, together with the type coercion information which happened during the unification process. A function *NewTagSymbol* is assumed which generates a fresh tag symbol each time it is invoked.

4. INTEGRATING INHERITANCE INTO PROLOG

4.1. A Simple Example

LOGIN is simply PROLOG where first-order terms are replaced by ψ-terms. Thus, we shall show that the skeleton of a PROLOG interpreter implementing a top-down, left-right backtracking search procedure can be adapted in a straightforward

Step	$\langle u, v\rangle$	Label	From step
1	$\langle X_0, Y_0\rangle$		
2	$\langle X_1, Y_1\rangle$	*advisor*	1
3	$\langle X_2, Y_2\rangle$	*roommate*	1
4	$\langle X_3, Y_4\rangle$	*secretary*	2
5	$\langle Y_0, Y_5\rangle$	*assistant*	2
6	$\langle Y_4, Y_2\rangle$	*representative*	3

FIGURE 7. Trace of the ψ-term unification of s and t.

Tag node	Successor	Subnodes	Carried subnodes
X_0	Y_0	*advisor* $\Rightarrow X_1$ *roommate* $\Rightarrow X_2$	
X_1	Y_1	*secretary* $\Rightarrow X_3$ *assistant* $\Rightarrow X_0$	
X_2	Y_2	*representative* $\Rightarrow X_3$	
X_3	Y_4		
Y_0	Y_5	*advisor* $\Rightarrow Y_1$ *roommate* $\Rightarrow Y_2$ *helper* $\Rightarrow Y_3$	
Y_1	$\dashv$	*secretary* $\Rightarrow Y_4$ *assistant* $\Rightarrow Y_5$	
Y_2	$\dashv$	*representative* $\Rightarrow Y_2$	
Y_3	$\dashv$	*spouse* $\Rightarrow Y_5$	
Y_4	Y_2		
Y_5	$\dashv$		*advisor* $\Rightarrow Y_1$ *roommate* $\Rightarrow Y_2$ *helper* $\Rightarrow Y_3$

FIGURE 8. Effect of *UNIFY*(s, t) before *REBUILD*.

New tag node	Class	Types	Conjoined type
Z_0	$\{X_0, Y_0, Y_5\}$	*student* *employee* *person*	*workstudy*
Z_1	$\{X_1, Y_1\}$	*faculty* f_1	f_1
Z_2	$\{X_2, Y_2, X_3, Y_4\}$	*employee* *student* *staff* *employee*	*workstudy*
Z_3	$\{Y_3\}$	w_1	w_1

FIGURE 9. Effect of *REBUILD* at the end of *UNIFY*(s, t).

manner.[6] The unification procedure is simply replaced by ψ-term unification, altered to allow for undoing coreference merging and type coercion upon backtracking.

Let us consider a simple example. We want to express 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; and 'C', 'D', 'F' are bad grades.

This information is depicted as the signature of Figure 10.

[6]For a formal "type-as-set" semantics of a typed logic of predicate with ψ-terms, see Appendix A.

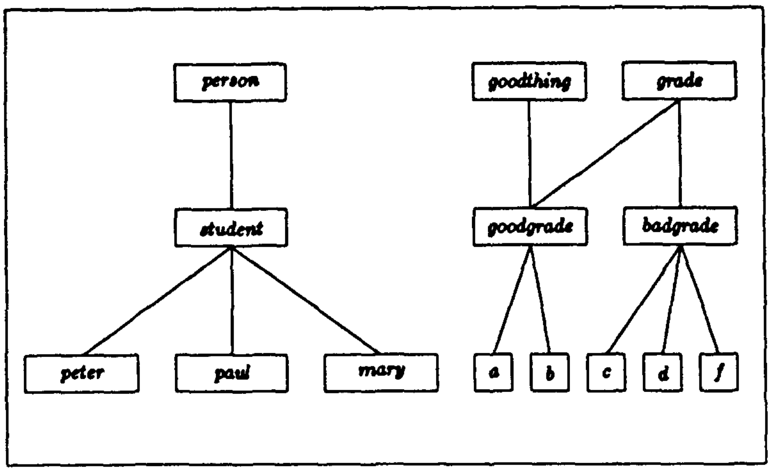

FIGURE 10. A signature for the simple example.

The above taxonomic information is expressed in LOGIN as

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

In this context, we 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 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 ; /* forces backtracking */
X = mary ; /* forces backtracking */
X = peter ;
No
```

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

$$likes(X\colon person, Y), got(X, Y). \tag{1}$$

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

$$got(X\colon person, X).$$

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

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

$$got(X\colon peter, Y\colon mary).$$

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

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

$$got(X\colon person, Y\colon goodthing).$$

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 $happy(X\colon \top)$, namely, the second rule of the *happy* procedure. The next resolvent is thus

$$likes(X\colon person, Y), got(Y, goodthing). \tag{2}$$

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

$$got(X\colon person, goodthing).$$

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

If backtracking is forced once again, resolvent (2) is restored. As seen before, establishing the first literal of this resolvent eventually leads to coercing $X\colon person$

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.

4.2. *The* LOGIN *Interpreter*

One important remark must be made in the light of the previous example. Namely, the conceptual PROLOG interpreter must be altered to accommodate undoing effects of ψ-term unification upon backtracking.

Now, ψ-term unification does not carry coercion information by composing variable substitutions as is done for conventional first-order terms. The bound unbound effect of the logic variable is here refined into gradual type coercion and coreference class merging. Hence, actual merging of classes as done in the ψ-term unification procedure by the *REBUILD* operation is out of the question, since it would make it very hard to "unmerge" classes after unification.

Although we could present an interpreter using the efficient way of doing *UNION* and *FIND* alluded to before, we wish to simplify our presentation by taking a simpler convention. Namely, the ψ-term unification algorithm used in this prototype of a LOGIN interpreter performs merging of two coreference classes by dereferencing a chronologically younger one toward an older one. This convention is common in existing PROLOG compiler implementations. Indeed, dereferencing from the younger to the older minimizes dereference chains, and suppresses the need (and overhead) of resetting older registers upon backtracking.

Although we do not have the experimental evidence to support a comparison of performance between these two ways of performing unification, we feel that our choice makes at least this initial description of a workable LOGIN interpreter easier to comprehend.

Be that as it may, it is not our intent to stress performance at this stage. Designing a LOGIN compiler will address all necessary issues regarding economy of work.

As before, we shall need some pseudo data structures to describe our algorithms. They are needed to represent a LOGIN program. Conceptually, we assume a function *PROGRAM* which takes as arguments a predicate symbol and its arity, and returns the associated definition, i.e., a list of clauses, or **nil** if none is defined for this pair. We shall assume a predefined linear list data type, with *head* and *tail* fields, and a list concatenation operation *APPEND*.

A *clause* consists of an array of ψ-terms which are the formal parameters for this alternative definition:

```
clause = record
            args  :  array of psiterm;
            body  :  list of literal
         end
```

A *definition* is the representation of a LOGIN procedure. It is essentially a linear list of alternative clauses. Hence, **nil** represents an empty definition:

```
definition = record
        firstdef  : clause;
        otherdef  : definition
      end
```

A *literal* is the atomic predicate/arity *cum* ψ-term arguments construct:

```
literal = record
        predicate  : symbol;
        arity      : integer;
        args       : array of psiterm
      end
```

As for ψ-terms, they are represented as *tagnode* before (this time without the superfluous *id*), augmented to allow for resetting upon backtracking. Namely, a field *newtype* will carry the coercion information, and a field *newsubnodes* the carried subterms. The reason for these fields is to preserve the original values of the type and subterms to be reset upon backtracking:

```
psiterm = record
        type         : constructor symbol;
        subnodes     : set of pairs ⟨label, psiterm⟩;
        coreference  : psiterm
        newtype      : constructor symbol;
        newsubnodes  : set of pairs ⟨label, psiterm⟩;
      end
```

Initial values for these extra fields are, respectively, **nil** (*coreference*), **nil** (*newtype*), and $\varnothing$ (*newsubnodes*). In order to reflect this modification in the representation of ψ-terms, we define the functions

$$\mathit{subnodes}(x) = x.\mathit{subnodes} \cup x.\mathit{newsubnodes}$$

and

$$\mathit{type}(x) = \begin{cases} x.\mathit{newtype} & \text{if } x.\mathit{newtype} \neq \mathbf{nil} \\ x.\mathit{type} & \text{otherwise} \end{cases}$$

and thus we must adapt

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

and

$$\mathit{subterm}(x, l) = y \qquad \text{such that} \quad \langle l, y \rangle \in \mathit{subnodes}(x).$$

The backbone of LOGIN is described in Figure 11 as the procedure *PROVE*. It works exactly as a PROLOG SLD-resolution interpreter does. The only difference is the *INHERIT* procedure which replaces first-order term unification, shown in Figure 12.

The reader will notice that the *INHERIT* procedure is a straightforward adaptation of the ψ-term unification procedure described in Figure 3.

```
function PROVE(resolvent: list of literal; background: trail)
returns: ⟨boolean; trail⟩;
  begin
  if resolvent = nil then return(⟨true, background⟩)
  else
    begin
    goal ← resolvent.head;
    nextgoals ← resolvent.tail;
    deflist ← PROGRAM(goal.predicate, goal.arity);
    success ← false;
    effect ← ∅;
    while (deflist ≠ nil) and (not success) do
      begin
      effect ← RESET(effect);
      currentdef ← COPY(deflist.firstdef);
      deflist ← deflist.otherdef;
      unifiable ← true; i ← 1;
      while unifiable and (i ≤ goal.arity) do
        begin
        ⟨unifiable, effect⟩ ← INHERIT(goal.args[i], currentdef.args[i], effect);
        i ← i + 1
        end;
      if unifiable
      then ⟨success, effect⟩ ← PROVE(APPEND(currentdef.body, nextgoals), effect)
      end;
    return(⟨success, effect ∪ background⟩)
    end
  end
```

FIGURE 11. The LOGIN Interpreter.

The function *COPY* takes a ψ-term and returns a new copy of it and its subterms. Because of our dereferencing convention, there is no more need for a separate *CARRYLABEL* function which tests how the dereference was performed by *UNION*. Also, the function *DEREF* is the same as *FIND* defined before, but without side effect.

The only information necessary for backtracking purposes is the set of ψ-term nodes affected during unification. We call such a set of ψ-terms a *trail*. The function *RESET* is applied to the current trail upon failure. It processes each of its ψ-terms by resetting the fields *coreference* and *newtype* to **nil**, and the field *newsubnodes* to $\emptyset$. The value returned by *RESET* is the empty set.

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

4.3. A More Complex Example

The example of Section 4.1 was simple, in the sense that it did not illustrate the use of inheritance among complex ψ-term objects—e.g., records with attributes. One such example is next described.

Consider the signature of Figure 6. As it stands, such a type symbol as *person* has virtually any possible attribute typed as $\top$. However, it may be desirable to

```
function $INHERIT(s, t: psiterm; background: trail)$;
returns : $\langle boolean; trail \rangle$;
  begin
  $PAIRS \leftarrow \{\langle s, t \rangle\}$;
  $unifiable \leftarrow$ true;
  $effect \leftarrow \emptyset$;
  while $unifiable$ and $(PAIRS \neq \emptyset)$ do
    begin
    remove $\langle x, y \rangle$ from $PAIRS$;
    $u \leftarrow DEREF(x)$;
    $v \leftarrow DEREF(y)$;
    if $u \neq v$ then
      begin
      $\sigma \leftarrow type(u) \wedge type(v)$;
      if $\sigma = \perp$ then $unifiable \leftarrow$ false
      else
        begin
        $v.coreference \leftarrow u$;
        $u.newtype \leftarrow \sigma$;
        $effect \leftarrow effect \cup \{u, v\}$;
        for each $l$ in $labels(v)$ do
          if $l$ in $labels(u)$
            then $PAIRS \leftarrow PAIRS \cup \{\langle subterm(u, l), subterm(v, l) \rangle\}$
            else $u.newsubnodes \leftarrow u.newsubnodes \cup \{\langle l, DEREF(subterm(v, l)) \rangle\}$
        end
      end
    end;
  return($\langle unifiable, effect \cup background \rangle$)
  end
```

FIGURE 12. The Inheritance Algorithm for LOGIN.

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 a string of characters between '000' and '999',
a field *middle* which must be a string of characters between '00' and '99', and
a field *last* which must be a string of characters between '0000' and '9999'.

We can express this in LOGIN as

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

where *name* is specified as

```
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 '$xy \ldots z$' 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 *student* is 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 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 of Figure 6. 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 *s*1 and *w*1.

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 than 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 are 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 20,000, 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 ψ-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'
```

5. CONCLUDING REMARKS

In the foregoing sections, we have presented 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. 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 the "set at a time" mode of computation essential to database applications.[7]

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 involved in the rules of a querying program.

APPENDIX A. A SEMANTICS OF INHERITANCE LOGIC

We give here (1) a "type-as-set" denotational semantics of the ψ-term calculus of partially ordered types, and (2) an interpretation of a first-order typed Horn-clause logic using ψ-terms as types.

A.1. Semantics of ψ-terms

We assume the existence of an abstract interpretation universe $\mathcal{U}$ of objects, where our types take meaning. A type is an intensional denotation of a set of elements in this universe. For example, the type *person* denotes the class of all objects in $\mathcal{U}$ which are categorized as persons. Some consensual agent is postulated—e.g., a programmer or an interpreter—for which such a categorization is meaningful. For example, it is reasonable to suppose that the reader's understanding of the English word "person" concurs with ours as far as our common-sense interpretation, namely, a particular *subclass* of the class $\mathcal{U}$ of *all* objects. Hence, in particular, the *least informative* type ($\top$) denotes the whole universe $\mathcal{U}$, and the *overdefined* type ($\bot$) is the inconsistent type, and denotes the empty set—the set of no object.

The subtype relation is interpreted as set inclusion in the semantic universe $\mathcal{U}$. For example, if the set of students is contained in the set of persons, then the type *student* is a subtype of the type *person*.

Let T be such a set of types, endowed with a subtype ordering relation $\leq$. A type semantics in an order homomorphism:

$$\iota : \langle T, \leq \rangle \to \langle 2^{\mathcal{U}}, \subseteq \rangle$$

where $2^{\mathcal{U}}$ is the set of all subsets of $\mathcal{U}$. Namely:

$$\iota[\![\top]\!] = \mathcal{U},$$

$$\iota[\![\bot]\!] = \varnothing, \qquad (3)$$

and for all s, t in T,

$$s \leq t \quad \Rightarrow \quad \iota[\![s]\!] \subseteq \iota[\![t]\!]. \qquad (4)$$

[7]See Appendix B.

Furthermore, if GLBs exist, it is desired that

$$\iota[\![s \wedge t]\!] = \iota[\![s]\!] \cap \iota[\![t]\!]. \tag{5}$$

In addition to signature type denotation, the information content of attributes and inheritance of attributes must be given a denotation which is congruent with the constructor types. For example, given that the type *person* is interpreted as a set of objects of $\mathcal{U}$, specifying the types of certain attributes of *person* is a means to denote a further restriction of the type *person*—e.g., talking about the class of persons whose last name is a character sting, rather than anything ($\top$). Thus, an attribute denotes the *intension* of a function between subsets of the universe $\mathcal{U}$. Attribute concatenation denotes function composition, and attribute coreference denotes the fact that certain functional diagrams commute.

More precisely, let Σ, $\leq$ be a partially ordered type signature, and let ι be a type semantics for it. Now, we need to indicate how to extend this type interpretation consistently to one for ψ-terms. Let us define a monoid homomorphism η from $\mathcal{L}^*$ with string concatenation to the set $\mathcal{U}^{\mathcal{U}}$ of functions from $\mathcal{U}$ to $\mathcal{U}$, with function composition. That is,

for each label l in $\mathcal{L}$, $\eta[\![l]\!]$ is a function in $\mathcal{U}^{\mathcal{U}}$;

$\eta[\![\varepsilon]\!]$ is the identity on $\mathcal{U}$;

$\forall u, v \in \mathcal{L}^* : \eta[\![u.v]\!] = \eta[\![v]\!] \circ \eta[\![u]\!]$.

The type semantics ι is extended to ψ-terms by the denotational semantic equations (6)–(9). These equations can be construed as "evaluation" rules for all possible syntactic cases. That is, the set which is the meaning of a given ψ-term is obtained by repeatedly applying Equations (6)–(9). These rules are clearly well founded (i.e., there cannot be an infinite interpretation sequence using them) because of the finiteness of a ψ-term's domain and coreference relation index. Also, the order in which these equations are applied does not matter, because of commutativity of set intersection.

Equation (6) treats the simple attribute case.

$$\iota[\![f(l \Rightarrow t)]\!] = \{ x \in \iota[\![f]\!] \mid \exists y \in \iota[\![t]\!],\ \eta[\![l]\!](x) = y \}. \tag{6}$$

It is now clear that the identification with $\perp$ of all ψ-terms where $\perp$ occurs is justified by this semantics. Indeed, by (6), it comes immediately that

$$\iota[\![f(l \Rightarrow \perp)]\!] = \iota[\![\perp]\!] = \varnothing .$$

Equation (6) is generalized to many attributes as follows:

$$\iota[\![f(l_1 \Rightarrow t_1; \ldots; l_n \Rightarrow t_n)]\!] = \bigcap_{i=1}^{n} \iota[\![f(l_i \Rightarrow t_i)]\!] \tag{7}$$

Attribute coreference means that compositions of attribute functions commute, as expressed by Equation (8):

$$\iota[\![\psi_1]\!] = \{ x \in \iota[\![\psi_2]\!] \mid \eta[\![l_0. \ldots .l_n]\!](x) = \eta[\![k_0. \ldots .k_m]\!](x) \}, \tag{8}$$

where

$$\begin{aligned}\psi_1 = f(&l_0 \Rightarrow g_1(l_1 \Rightarrow \cdots g_n(l_n \Rightarrow X : t) \ldots); \\ &k_0 \Rightarrow h_1(k_1 \Rightarrow \cdots h_m(k_m \Rightarrow X) \ldots))\end{aligned}$$

and

$$\psi_2 = f(l_0 \Rightarrow g_1(l_1 \Rightarrow \cdots g_n(l_n \Rightarrow t)\ldots);$$
$$k_0 \Rightarrow h_1(k_1 \Rightarrow \cdots h_m(k_m \Rightarrow t)\ldots)).$$

Finally, cyclic coreference corresponds to fixed points of attribute functions, as expressed in

$$\iota[\![\psi_1]\!] = \{x \in \iota[\![\psi_2]\!] | \eta[\![l_0. \ldots .l_n]\!](x) = x\}, \tag{9}$$

where

$$\psi_1 = X: f_1(l_1 \Rightarrow \cdots f_n(l_n \Rightarrow X))$$

and

$$\psi_2 = f_1(l_1 \Rightarrow \cdots f_n(l_n \Rightarrow f_1)).$$

It is not difficult to verify that axioms (3)–(5) hold for this type semantics.

A.2. Typed Horn Clauses

It is clear how a typed logic may be mapped into an untyped logic. Namely, any well-formed quantified logical formula of the form:

$$\forall x : t . \alpha$$

is semantically equivalent to

$$\forall x . \mathbf{1}_t(x) \Rightarrow \alpha,$$

where $\mathbf{1}_t$ is the characteristic predicate of the set denoted by the type t (i.e., $\mathbf{1}_t(x)$ is true if and only if x is an element of the set denoted by t). Similarly for existential quantifiers:

$$\exists x : t . \alpha$$

is semantically equivalent to

$$\exists x . \mathbf{1}_t(x) \Rightarrow \alpha.$$

A PROLOG program is a conjunction of Horn clauses of the form

$$\alpha_0 \Rightarrow \alpha_1, \ldots, \alpha_n$$

where the syntactic scope of a variable is limited to a single clause, such that the following implicit quantification rules apply:[8]

all variables which appear on the left side of the symbol '$\leftarrow$' are universally quantified, and

all other variables (i.e., all those which appear in the right-hand side but not in the left-hand side) are existentially quantified.

For example the Horn clause

$$p(X,Y) \leftarrow q(X,Z), r(Z,Y)$$

reads

$$\forall X, \forall Y, \exists Z . q(X,Z) \& r(Z,Y) \Rightarrow p(X,Y).$$

[8]Naturally, in prolog clauses seen in prenex disjunctive normal form, *all* variables are universally quantified. However, this remark deals with a PROLOG clause seen as an implication where each implicit quantifier would be as close to its variable as possible.

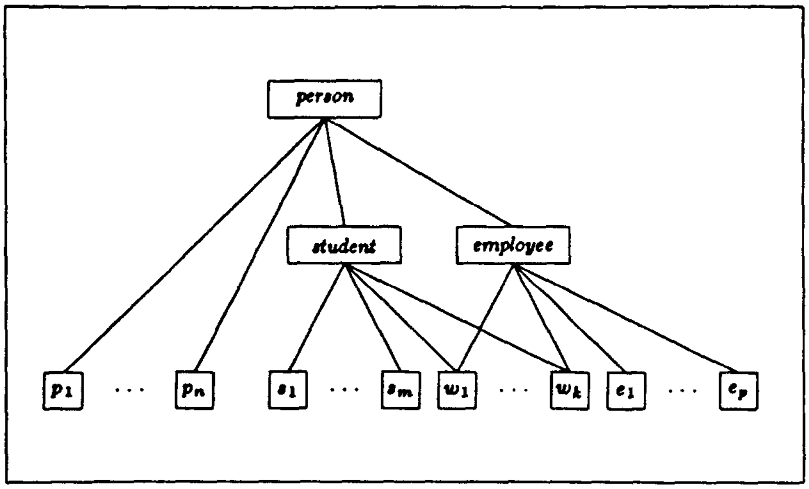

FIGURE 13. A signature which is not a semilattice.

Thus, LOGIN may be similarly interpreted as a typed extension of Horn logic using the semantics of types described in Section A.1.

APPENDIX B. A SIMPLE SEMILATTICE CONSTRUCTION

The ψ-term unification algorithm given in Section 3.3 relies on the assumption that the signature Σ must be a lower semi-lattice; i.e., that a *unique* GLB exists in Σ for any two type symbols in Σ (given by the $\wedge$ operation). However, in practice this is not quite a reasonable assumption to make. Indeed, in order to maintain this assumption, as the size of type signature grows, there must be specified an exponential number of pairwise GLBs—clearly, an inappropriate demand on a programmer.

Instead, it would be simpler to embed a partially ordered type signature Σ which is not a lower semilattice into the least such structure which would contains it—up to some isomorphism. This embedding must preserve the order structure of Σ, and in particular, existing GLBs. Such an embedding must also be semantically sound.

The idea is rather simple, and makes intuitive sense. Let us consider for example the signature of Figure 13. Types $w_1, \ldots, w_k$ are both students and employees. However, there is not a common type symbol to designate the set of students and employees. Thus, the GLB of *student* and *employee* in this signature cannot be defined as a unique element of the signature. Nevertheless, it makes sense to say that the GLB of *student* and *employee* ought to be the *set* $\{w_1, \ldots, w_k\}$. This is precisely the effect that the following construction achieves.

In what follows, we make the assumption that the signature is finite.[9] First, we need some definitions.

The *restricted powerset* of a partially ordered set S, $\leq$ (*poset*) is the set $\mathbf{2}^{(S)}$ of nonempty finite subsets of pairwise *incomparable* elements of S. Such subsets are

[9]In fact, as shown in [2], such a construction can be performed for an infinite signature which is Noetherian—i.e., one which does not contain infinite ascending chains.

called *crowns*, and are partially ordered by a relation $\sqsubseteq$ defined by

$$X \sqsubseteq Y \quad \text{iff} \quad \forall x \in X, \exists y \in Y, x \leq y.$$

Given a poset S, $\leq$, the *canonical injection* from S into $\mathbf{2}^{(S)}$ is the function which takes every element x of S to the singleton $\{x\}$. This simple function has the nice property that

$$\forall x \in S, \quad \forall y \in S, \quad \{x\} \sqsubseteq \{y\} \text{ iff } x \leq y.$$

That is, it is an order homomorphism.

Given any subset X of S, we define its *maximal restriction* $\lceil X \rceil$ as the set of maximal elements of X. Clearly, $\lceil X \rceil$ is in $\mathbf{2}^{(S)}$, and defined for all subsets of a finite poset S.

Given some element x of S, we denote by $\underline{x}$ the subset of S of all lower bounds of x. That is,

$$\underline{x} = \{ y \in S | y \leq x \}.$$

Then, for any two elements a, b in S, $\underline{a} \cap \underline{b}$ is the set of common lower bounds of a and b in S.

Finally, the following operation $\sqcap$ can be defined for any pair of subsets X, Y in $\mathbf{2}^{(S)}$:

$$X \sqcap Y = \left\lceil \bigcup_{\substack{a \in X \\ b \in Y}} \underline{a} \cap \underline{b} \right\rceil, \tag{10}$$

and this operation is a GLB operation in $\mathbf{2}^{(S)}$. Informally, we may describe what is performed by the operation of Equation (10) as "skimming the cream off the crown" of the set of all common lower bounds of all pairs of elements.

As a result, $\mathbf{2}^{(S)}$, $\sqsubseteq$, $\sqcap$ is a lower semilattice. Furthermore, we observe that if two elements x and y in S already have a unique GLB z in S, it follows that

$$\{x\} \sqcap \{y\} = \{z\}.$$

Hence, this construction is a structure embedding, since it preserves the ordering and the GLBs when they exist in S.

Now, we are justified in taking the liberty of writing simply x rather than $\{x\}$ for any single element of a type signature Σ, and extending the signature to $\mathbf{2}^{(\Sigma)}$, the GLB-preserving lower-semilattice extension of Σ. And this is the "least" such possible structure, because if Σ is already a lower semilattice, then it is isomorphic to its canonical injection into $\mathbf{2}^{(\Sigma)}$.

That such an embedding is semantically sound becomes clear when one understands the semantics of a type such as $\{t_1, \ldots, t_n\}$. In a "type as set" semantics, the LUB of the types t_i $(i = 1, \ldots, n)$ should also be the LUB of everything that is subsumed by every t_i. Hence, it results naturally that

$$\iota[\![\{t_1, \ldots, t_n\}]\!] = \bigcup_{i=1}^{n} \iota[\![t_i]\!].$$

In the implementation of an operation such as defined by Equation (10), maximal common lower bounds of two elements can be computed by an asynchronous parallel marking method. This achieves efficient "set at a time" computation, as opposed to the conventional "element at a time" way of PROLOG.

We wish to thank Scott Danforth, Matthias Felleisen, and Joe Scullion for their sharp proofreading and corrections of an earlier draft. We are especially indebted to Matthias for pointing out technical inconsistencies, for challenging discussions, and for his enthusiastic encouragements. We are also grateful to Bob Boyer, Fernando Pereira, and David Russinoff for helping us eliminate several misconceptions. Finally, the form and content of this paper have benefited from several conversations with François Banchilhon, Shalom Tsur, and Carlo Zaniolo.

REFERENCES

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, Univ. of Pennsylvania, Philadelphia, 1984.
3. Allen, J. F., and Frish, A. M., What's in a Semantic Network, in: *Proceedings of the 20th Annual ACL Meeting*, Assoc. for Computational Linguistics, 1982.
4. Brachman, R. J., Fikes, R. E., and Levesque, H. J., KRYPTON: A Functional Approach to Knowledge Representation, FLAIR Technical Report No. 16, Fairchild Lab. for Artificial Intelligence Research, Fairchild Research Center, Palo Alto, Calif., May 1983.
5. Deliyanni, A. and Kowalski, R. A., Logic and Semantic Networks, *Comm. ACM* 22(3):184–192 (1979).
6. Huet, G., Résolution d'Equations dans des Langages d'Ordre 1,2,...,ω, Thèse de Doctorat d'Etat, Univ. de Paris VII, France, 1976.
7. Kowalski, R. A., Logic for Data Description, in: H. Gallaire and J. Minker (eds.), *Logic and Data Bases*, Plenum 1978, pp. 77–103.
8. McSkimin, J. R. and Minker, J., A Predicate Calculus Based Semantic Network for Question-Answering Systems, in: N. Findler (ed.), *Associative Networks—The Representation and Use of Knowledge by Computers*, Academic, New York, 1979.
9. Reynolds, J. C., Transformational Systems and the Algebraic Structure of Atomic Formulas, In: D. Michie (ed.), *Machine Intelligence 5*, Edinburgh. U.P., 1970.
10. Robinson, J. A., A Machine-Oriented Logic Based on the Resolution Principle, *J. Assoc. Comput. Mach.* 12(1):23–41 (1965).