

A Database Interface for Complex Objects

Marcel Holsheimer[†]

Centrum voor Wiskunde en Informatica (CWI)

Postbus 4079

1009 AB Amsterdam

The Netherlands

marcel@cwi.nl

Rolf A. de By

Computer Science Department

University of Twente

P.O. Box 217

7500 AE Enschede

The Netherlands

deby@cs.utwente.nl

Hassan Aït-Kaci[‡]

School of Computing Science

Simon Fraser University

Burnaby, British Columbia

V5A 1S6, Canada

hak@cs.sfu.ca

Abstract

We describe a formal design for a logical query language using ψ -terms as data structures to interact effectively and efficiently with a relational database. The structure of ψ -terms provides an adequate representation for so-called complex objects. They generalize conventional terms used in logic programming: they are sorted attributed structures, ordered thanks to a subsort ordering. Unification of ψ -terms is an effective means for integrating multiple inheritance and partial information into a deduction process. We define a compact database representation for ψ -terms, representing part of the subsorting relation in the database as well. We describe a retrieval algorithm based on an abstract interpretation of the ψ -term unification process and prove its formal correctness. This algorithm is efficient in that it incrementally retrieves only additional facts that are actually needed by a query, and never retrieves the same fact twice.

The difficulty lay in the form and economy of it,
so to dispose such a multitude of materials as not
to make a confused heap of incoherent parts but
one consistent whole.

EPHRAIM CHAMBERS, *Cyclopaedia*

[†] Work done at University of Twente.

[‡] Work done at Digital's Paris Research Laboratory.

1 Introduction

1.1 Motivation and contribution

The combination of logic programming languages and database systems has been a research theme for the last decade in both logic programming and database communities. The interest from a logic programming perspective came when the need was felt for manipulating large sets of facts. Usually Prolog was coupled with a relational database. In [9], Ceri *et al.* provide an excellent overview of work in this area. In the database community, it was felt that the logic programming paradigm offers interesting opportunities as a database query language. This resulted in logical query languages like \mathcal{LDL} [14] and NAIL! [13].

So-called complex objects have recently been studied for use in database systems [7, 8]. Much of what has been proposed in those studies is derived from earlier work extending first-order terms to ψ -terms [1]. The latter notion has had a more direct application in programming language design [4, 2, 6] than in database systems. Still, the functionality and naturalness of deductive queries over ψ -terms is a strong motivation for providing a logic programming language using ψ -terms with an effective means to access large volumes of data and knowledge stored in a database (see [5] for a convincing example).

We propose a formal design for an effective coupling of such a language with a relational database. For the purpose of our presentation and experimentation, we use the specific language LIFE [2], but this implies no loss of generality. Indeed, although we formulate it using ψ -terms, our design is directly applicable to any logical query language with complex objects represented as Prolog terms or as data structures *à la* [7, 8], since all these models turn out to be special cases of ψ -terms. We present the theoretical view of our proposed database support of that language and discuss the results. Our theoretical design was put into practice as the basis of an experimental implementation [12].

Although our experiment may be categorized as providing database support to a logic programming language, it goes beyond previous research in that it considers a language with sorts and attributed terms, which can be arbitrarily nested, and provide multiple inheritance. As will be shown, due to the specific characteristics of LIFE's sort system, our experiment has yielded a form of database support that not only allows querying for facts, but also posing abstract queries, that is, queries that ask for general knowledge as opposed to factual knowledge.

1.2 Organization of paper

Before we delve into technicalities, here is a brief introductory overview of the paper. Our system is organized as sketched in Figure 1 and consists of three subsystems; namely, the LIFE system, an interface written in LIFE, and an external relational database. The coupled system is intended to represent the facts of LIFE in the database and to retrieve these facts, when needed by the LIFE system.

Hence, the functionality of the interface is twofold. Firstly, it provides a compact database representation for logical facts. As we shall see in Section 2, these facts are ordered by a subsumption relation induced by a subsort ordering

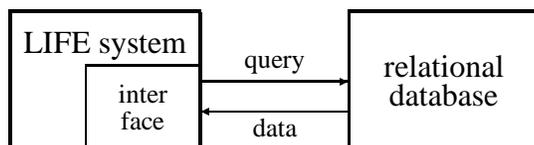


Figure 1. Architecture of the system.

on functors. In Section 3, we propose to group facts into what we call *qualified segments*, such that the subsort relationships involving symbols in these facts are implicitly represented. We also compress segments before storage in the database.

Secondly, for the retrieval of facts, we use a *tight coupling* [15, 16], where facts are loaded when needed by the LIFE system. In Section 4, we describe an abstraction of the unification process, where qualified segments in the database are approximated by a set of generalizations, called *qualifier*. If facts from the database are requested, we use the qualifier and the current goal, a term, to construct a *candidate*: a selection condition on the segment, retrieving all facts that unify with this goal. In Section 5, we show that not all subsort relationships need to be stored in the LIFE-system, since some are implicitly represented in the database. In Section 6, we optimize the retrieval process, by storing loaded facts in the internal database and retrieving each fact only once. We conclude with Section 7, with a recapitulation of our work and a brief overview of the perspectives it offers. No particular background is required to understand the technical contents of this paper other than elementary discrete algebra, shreds of logic programming, and basic notions of relational and deductive databases.

2 The facts of LIFE

LIFE (*Logic, Inheritance, Functions, Equations*) is a logic programming language extending Prolog terms as described in [2, 4, 6]. The user can specify inclusion relationships between functor symbols, thus enabling the direct representation and use of taxonomic information. Thus, functors are called sorts and no longer differentiated from values. For example, we can state that *apples* is a subsort of *food*, so that a fact *likes(mary,food)*, stating that mary likes food, implies that mary likes apples as well.

To make use of a subsorting relation in a logic programming language, the unification operation must be redefined. The subsorting relation generates a partial order on the set of all terms called *term subsumption*. Unification of two terms computes their *greatest lower bound* (GLB) with respect to term subsumption. Failure of unification is denoted by a special term: the symbol \perp (*'bottom'*).

For the purpose of our presentation, it will suffice to assume that a LIFE program P consists of the specification of the subsort ordering, and logical rules in the form of Horn-clauses. The essential point to keep in mind is that the

literals making up a program's clauses are ψ -terms rather than conventional Prolog terms. Hence, as is the case in deductive database languages, the Horn clauses are separated into the *extensional* database (EDB)---i.e., the facts containing no variables---and the *intensional* database (IDB)---the rest.

Our idea is to represent the (presumably numerous) facts of a LIFE program's EDB as flat relations to store in an external relational database. Then, designing an interface amounts to defining an intermediate representation allowing to translate from facts of LIFE (i.e., ψ -terms) to database tuples and back. To be correct, a database retrieval algorithm responding to a LIFE query through this interface must be sound (i.e., retrieve *no irrelevant* tuples) and complete (i.e., retrieve *all relevant* tuples). Hence, the interface design and the correctness of retrieval depend in some essential way on the formalization of ψ -terms. This section is meant to give all the preliminary formalities that we use, introducing basic and disjunctive ψ -terms, sort signatures, subsumption, and related notions. From this point on, whenever we say "term" we shall mean (possibly disjunctive) " ψ -term."¹

2.1 Terms

A *basic* term is built out of *sort symbols* and *attribute labels*. Let \mathcal{L} be the set of all attribute labels, and \mathcal{S} the set of all sort symbols, including \top ("top") and \perp ("bottom").

Definition 1 (Basic term) A basic term p is an expression of the form:

$$s(l_1 \Rightarrow p_1, \dots, l_n \Rightarrow p_n), \quad n \geq 0,$$

where:

- $s \in \mathcal{S}$ is the root symbol of p , denoted by $\mathbf{root}(p)$.
- $l_1, \dots, l_n \in \mathcal{L}$ are pairwise distinct attribute labels.
- p_1, \dots, p_n are terms: the subterms of p .

If $n = 0$, p is said to be *atomic*, and simply written as s . Otherwise, p is said to be *attributed*. The attribute-subterm list is unordered. A term with at least one occurrence of the symbol \perp is considered to be equal to the term \perp . We call Ψ the set of all basic terms that can be constructed from sort symbols in \mathcal{S} and labels in \mathcal{L} .

Example 2.1 An example of a basic term is:

$$\begin{aligned} & \text{likes}(\text{who} \Rightarrow \text{mary}, \\ & \quad \text{born} \Rightarrow \text{date}(\text{day} \Rightarrow 24, \\ & \quad \quad \text{month} \Rightarrow \text{january}, \\ & \quad \quad \text{year} \Rightarrow 1965), \\ & \quad \text{what} \Rightarrow \text{apples}). \end{aligned}$$

¹More precisely, we shall mean ψ -terms *without variables* since only EDB facts will be considered.

The root symbol is *likes*; it has three subterms with attribute labels *who*, *born* and *what*. The sort symbols are *likes*, *mary*, *date*, *24*, *january*, *1965*, and *apples*. The attribute labels are *who*, *born*, *day*, *month*, *year*, and *what*. \square

We shall use a more convenient mathematical characterization of a basic term that is formally equivalent to their syntactic representation of Definition 1. It sees a term as a mapping from a set of *occurrences* (i.e., strings of labels in the free monoid \mathcal{L}^*) to \mathcal{S} , assigning sort symbols to each of these occurrences.

Definition 2 (Occurrence) *An occurrence is a string formed by concatenating labels, separated by ‘.’. The root label is denoted by the empty string ε . The set of all occurrences \mathcal{L}^* is inductively defined as $\mathcal{L}^* := \varepsilon \mid \mathcal{L}.\mathcal{L}^*$, where $a.\varepsilon = \varepsilon.a = a$ for any occurrence a .*

In what follows, every time we refer to term p , we mean the generic one in Definition 1.

Definition 3 (Occurrence domain) *The set of occurrences actually appearing in a term p is the occurrence domain Δ_p : the smallest subset of \mathcal{L}^* for which:*

- $\varepsilon \in \Delta_p$ and
- $l_i.a \in \Delta_p$ iff l_i is the label in p denoting the subterm p_i , and $a \in \Delta_{p_i}$.

Definition 4 (Sort function) *To each term p there corresponds a sort function $\psi_p : \mathcal{L}^* \rightarrow \mathcal{S}$ which assigns a sort symbol to each occurrence:*

$$\psi_p(a) = \begin{cases} \top & \text{if } a \notin \Delta_p \\ \mathbf{root}(p) & \text{if } a = \varepsilon \\ \psi_{p_i}(a') & \text{if } a = l_i.a' \end{cases}$$

Hence, a basic term is formally characterized as a pair $p = \langle \Delta_p, \psi_p \rangle$.

Example 2.2 Referring to the term in Example 2.1, the domain is $\{\varepsilon, \textit{who}, \textit{born}, \textit{born.day}, \textit{born.month}, \textit{born.year}, \textit{what}\}$. The sort function is defined as: $\psi(\varepsilon) = \textit{likes}$, $\psi(\textit{who}) = \textit{mary}$, $\psi(\textit{born}) = \textit{date}$, $\psi(\textit{born.day}) = 24$, etc. Note that the sort function returns the \top -symbol for any occurrence not in the occurrence domain, for example $\psi(\textit{day.what}) = \top$. \square

2.2 A short terminological digression

For the sake of self-containment and to settle some terminology, we indulge in a brief *intermezzo* defining a few general basic order-theoretic notions that we shall use in the rest of this paper. All definitions in this short digression will refer to a partially-ordered set, or *poset*, $\langle S, \leq \rangle$.

Recall that a chain of S is a totally ordered subset of S . Let us also recall the notion of *cochain*, a dual of the more familiar notion of chain:

Definition 5 (Cochain) *A cochain C of S is a subset of S where all distinct elements are mutually incomparable. Formally, $C \times C \cap \leq = \mathbf{1}_C$.²*

²Where $\mathbf{1}_X = \{(x, x) \mid x \in X\}$ is the identity relation on X .

The set of all cochains of S is denoted as $\mathbf{coc}(S)$. The set $\mathbf{coc}(S)$ is itself partially ordered as follows.

Definition 6 (Cochain ordering) $\forall C_1, C_2 \in \mathbf{coc}(S), C_1 \sqsubseteq C_2$ iff $\forall x_1 \in C_1, \exists x_2 \in C_2 : x_1 \leq x_2$.

Note that the empty set \emptyset is a cochain. In particular, the empty set is the *least* element in $\mathbf{coc}(S)$; that is, $\forall C \subseteq S : \emptyset \sqsubseteq C$.

Note also that singletons of elements of S are cochains too. In fact, the cochain ordering \sqsubseteq coincides with \leq on singletons; namely, $\forall x, x' \in S : \{x\} \sqsubseteq \{x'\}$ iff $x \leq x'$. For this reason, an element x of S may be identified with the singleton $\{x\}$. Hence, the cochain ordering \sqsubseteq is a “natural” extension of the base ordering \leq and so we shall use only one symbol (\leq) indifferently on base elements or cochains of S without risk of confusion.

It will be convenient to refer, for a given element of S , to specific subsets of its upper bounds or lower bounds. The following definitions introduce a few that we will use. In what follows, x and x' denote elements of such a set S .

Definition 7 (Ancestors) *The set of ancestors of x is the set $\mathbf{anc}(x)$ of elements greater than, or equal to x :*

$$\mathbf{anc}(x) = \{x' \in S \mid x \leq x'\}.$$

Definition 8 (Descendants) *The set of descendants of x is the set $\mathbf{des}(x)$ of elements smaller than, or equal to x :*

$$\mathbf{des}(x) = \{x' \in S \mid x' \leq x\}.$$

Given $S' \subseteq S$, let $\lceil S' \rceil$ (resp., $\lfloor S' \rfloor$) denote the set of all its maximal (resp., minimal) elements.³ We define *parents* and *children*, as well as *maximal common lower bounds* and *minimal common upper bounds*, in terms of ancestors and descendants as follows.

Definition 9 (Parents and children) *The parents of x are its immediate upper bounds; i.e., the minimal ancestors, excluding x itself:*

$$\mathbf{par}(x) = \lfloor \mathbf{anc}(x) \setminus \{x\} \rfloor$$

Dually, the children of x are its immediate lower bounds; i.e.,

$$\mathbf{chi}(x) = \lceil \mathbf{des}(x) \setminus \{x\} \rceil$$

Definition 10 (Maximal common lower bounds) *The set of maximal common lower bounds of x and x' is denoted as $x \sqcap x'$, and defined as:*

$$x \sqcap x' = \lceil \mathbf{des}(x) \cap \mathbf{des}(x') \rceil.$$

³To be well-defined, this requires that S not contain infinitely ascending (resp., descending) chain. So we shall implicitly assume this. In fact, all the posets on which we will use these operations will be finite.

Definition 11 (Minimal common upper bounds) Dually, the set of minimal common upper bounds of s and s' is denoted $x \sqcup x'$, and defined as:

$$x \sqcup x' = \left[\mathbf{anc}(x) \cap \mathbf{anc}(x') \right].$$

Note that all the sets introduced by the four previous definitions are cochains.

Finally, given two functions f and f' from a set A to a poset $\langle \mathcal{S}, \leq \rangle$, we say that $f \leq f'$ whenever $\forall a \in A : f(a) \leq f'(a)$.

This concludes our terminological digression. We now return to our topical considerations.

2.3 Sort signature

The set of sort symbols \mathcal{S} comes with a subsort ordering \leq . The set \mathcal{S} and the ordering form a *sort signature*, a poset $\Sigma = \langle \mathcal{S}, \leq \rangle$. We may assume the sort signature to be fixed.

Definition 12 (Sort signature) A sort signature Σ is a poset $\langle \mathcal{S}, \leq \rangle$, where:

- \mathcal{S} is the set of sort symbols, containing top symbol \top and bottom symbol \perp .
- $\leq \subseteq \mathcal{S} \times \mathcal{S}$ is a partial order---the subsorting---on \mathcal{S} such that $\forall s \in \mathcal{S} : \perp \leq s \leq \top$.

Example 2.3 All the examples to follow will use a sort signature consisting of a set $\mathcal{S} = \{\top, \perp, \textit{student}, \textit{emp}, \textit{mary}, \textit{likes}, \textit{food}, \textit{apples}, \textit{sweets}, \textit{cookies}, \textit{chocolate}\}$, and for subsorting the least ordering such that $\textit{apples} \leq \textit{food}$, $\textit{sweets} \leq \textit{food}$, $\textit{cookies} \leq \textit{sweets}$ and $\textit{chocolate} \leq \textit{sweets}$, expressing that apples and sweets are food, and cookies and chocolate are sweets; and such that $\textit{mary} \leq \textit{student}$ and $\textit{mary} \leq \textit{emp}$, expressing that mary is both a student and an employee. This sort signature will be referred to as Σ and is depicted in Figure 2. \square

2.4 Term subsumption

The partial order \leq on sort symbols extends to the set of all terms as follows:

Definition 13 (Basic term subsumption) The basic term subsumption relation \triangleleft on the set of all basic terms Ψ is defined as $p \triangleleft p'$ iff $p = \perp$ or $\psi_p \leq \psi_{p'}$.

Example 2.4 The term $p_1 = \textit{likes}(\textit{who} \Rightarrow \textit{mary}, \textit{what} \Rightarrow \textit{apples})$ is subsumed by the term $p_2 = \textit{likes}(\textit{who} \Rightarrow \textit{mary}, \textit{what} \Rightarrow \textit{food})$ since $\textit{apples} \leq \textit{food}$. Term p_1 is also subsumed by the term $p_3 = \textit{likes}(\textit{who} \Rightarrow \textit{mary})$ since the sort symbol is \top for any occurrence that is not in the occurrence domain; i.e., $\psi_{p_1}(\textit{what}) = \textit{apples} \leq \psi_{p_3}(\textit{what}) = \top$. Thus any basic term is subsumed by \top and subsumes \perp . \square

Note since \mathcal{S} is a subset of Ψ , \triangleleft coincides with \leq on it. Therefore, \triangleleft can be seen as a “natural” extension of the subsort ordering \leq and therefore we shall again use only one symbol (\leq) indifferently on sort symbols or basic terms without risk of confusion.

As expected, we now extend terms to cochains of terms.

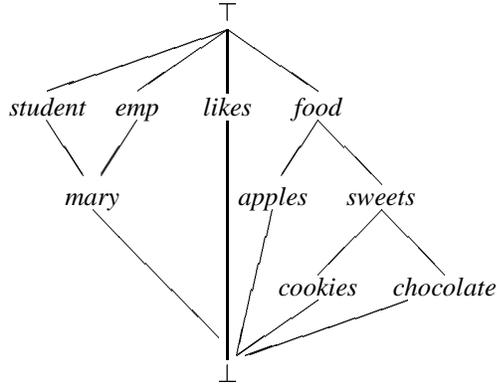


Figure 2. The sort signature Σ .

Definition 14 (Disjunctive terms) A disjunctive term is a cochain of basic terms.

Term subsumption is naturally extended to disjunctive terms as the cochain ordering of basic term subsumption. Hence, by “term” we now shall mean basic or possibly disjunctive term.

As usual, a singleton disjunctive term $\{p\}$ is identified with the basic term p . In particular, the singleton set $\{\top\}$ is identified with the basic term \top . This is natural since they are both greatest elements for term subsumption. Similarly, $\{\perp\}$ is identified with the basic term \perp . Note that this is natural since they are both least elements. However, the empty set \emptyset is also the least element of $\mathbf{coc}(\Psi)$, and hence we can identify all three: $\perp = \{\perp\} = \emptyset$.

The following is a particular case of a more general result in [1].

Theorem 1 The poset $\langle \mathbf{coc}(\Psi), \leq \rangle$ is a lattice.⁴

Proof Greatest lower bounds are constructed as follows. For basic terms p and p' , the (possibly disjunctive) term $p \wedge p'$ is the set of maximal elements of the set of all basic terms $u = \langle \Delta_u, \psi_u \rangle$ such that:

- $\Delta_u = \Delta_p \cup \Delta_{p'}$,
- $\forall a \in \Delta_u : \psi_u(a) \in \psi_p(a) \sqcap \psi_{p'}(a)$.

For (possibly singleton) disjunctive terms C, C' , it is given by $C \wedge C' = [\{p \wedge p' \mid p \in C, p' \in C'\}]$.

Dually, least upper bounds (LUB) are constructed as follows. For basic terms p and p' , the (possibly disjunctive) term $p \vee p'$ is the set of minimal elements of the set of all basic terms $u = \langle \Delta_u, \psi_u \rangle$ such that:

⁴Recall that a lattice L is a poset where a unique greatest lower bound and a unique least upper bound both exist in L for any finite non-empty subset of L .

- $\Delta_u = \Delta_p \cap \Delta_{p'}$,
- $\forall a \in \Delta_u : \psi_u(a) \in \psi_p(a) \sqcup \psi_{p'}(a)$.

For (possibly singleton) disjunctive terms C, C' , it is given by $C \vee C' = \lfloor \{p \vee p' \mid p \in C, p' \in C'\} \rfloor$.

It is easy to verify that these operations are lattice operations with respect to term subsumption. ■

Note that if the sort signature Σ is a lattice, then so is Ψ , and moreover, it is then a sublattice of $\mathbf{coc}(\Psi)$.

Example 2.5 The GLB of terms p_1 and p_2 in Example 2.4 is p_1 , since $p_1 \leq p_2$. The GLB of $\mathit{likes}(\mathit{who} \Rightarrow \mathit{student})$ and $\mathit{likes}(\mathit{who} \Rightarrow \mathit{emp})$ is $\mathit{likes}(\mathit{who} \Rightarrow \mathit{mary})$. Their LUB is $\mathit{likes}(\mathit{who} \Rightarrow \top)$. The GLB of atomic terms food and $\mathit{student}$ is \perp ; *i.e.*, we cannot unify these. □

3 Representation in a database

We now discuss the storage of facts in an external relational database.

3.1 Qualified segments

In a relational database, identically formed objects are grouped together in a relation. We must define a similar grouping on facts that we store in the external database. We must also find a way to represent subsort information relevant to sort symbols in these facts in the database as well as there is no evident way to express subsumption in relational algebra. Therefore, if a fact is stored in a database relation, it should imply that particular subsort relationships are defined for symbols in this fact. Thus we should group facts with similar subsort relationships for its symbols, for example symbols with the same parents or children or both. However, there is a trade-off: the more subsort information is implicitly represented, the more database relations are needed to store all facts.

We choose to group facts with the *same set of parents* for all symbols at each given occurrence. It turns out that this is a natural choice since sharing parents is the most immediate commonality, akin to values being of the same sort. These sets are called *qualified segments*:

Definition 15 (Qualified segment) *A qualified segment Q is a set of non-bottom facts such that all facts have the same set of parents for the sort symbol at each occurrence:*

$$\forall f, f' \in Q, \forall a \in \Delta_f : \mathbf{par}(\psi_f(a)) = \mathbf{par}(\psi_{f'}(a))$$

With some easy thinking, one can convince oneself that all facts in Q must necessarily be identically formed. Indeed, the occurrence domain is the same for all facts in a qualified segment, since parents are the same for symbols at each occurrence. For a qualified segment Q , the common occurrence domain of all facts is denoted Δ_Q .

For a program P , we can use multiple qualified segments to store part of the facts in P in the database. We store each qualified segment in a separate database relation, and in the interface we store a description of the contents of each segment, called the *qualifier*. A qualifier is a set of terms, that are generalizations of all facts in the qualified segment:

Definition 16 (Qualifier) *To a qualified segment Q corresponds a qualifier, denoted $\mathbf{qua}(Q)$, which is the LUB of all facts in Q .*

Example 3.1 Let us assume the two facts of LIFE $\mathit{likes}(\mathit{who} \Rightarrow \mathit{mary}, \mathit{what} \Rightarrow \mathit{sweets})$ and $\mathit{likes}(\mathit{who} \Rightarrow \mathit{mary}, \mathit{what} \Rightarrow \mathit{apples})$. Since both facts have the same parents for all sort symbols, we can represent them in a qualified segment $Q = \{\mathit{likes}(\mathit{who} \Rightarrow \mathit{mary}, \mathit{what} \Rightarrow \mathit{sweets}), \mathit{likes}(\mathit{who} \Rightarrow \mathit{mary}, \mathit{what} \Rightarrow \mathit{apples})\}$. The qualifier is $\mathbf{qua}(Q) = \mathit{likes}(\mathit{who} \Rightarrow \mathit{mary}, \mathit{what} \Rightarrow \mathit{food})$. \square

An important remark is that the qualifier of a qualified segment is always a *strict* generalizer of all facts of the segment. This is a consequence of having grouped facts in the same qualified segment if and only if the sort symbols at all their occurrences shared the same parents.⁵ And thus, as we will see in Section 5, a qualifier and the terms in the corresponding segment, implicitly represent subsort relationships.

3.2 Database relations

A relational database consists of database relations:

Definition 17 (Database relation) *A database relation R_T is a set $\{r_1, r_2, \dots, r_m\}$, ($m \geq 0$) of n -ary tuples ($n \geq 1$) and is identified by its relation name R and a set of attribute names $T = \{t_1, t_2, \dots, t_n\}$. For a particular tuple r , the value of attribute t is denoted as $r.t$.*

We store a qualified segment Q in database relation R_T by representing each fact in Q as a tuple in R_T . We represent fact f as a tuple r by *flattening* the fact; *i.e.*, we define a bijective function v ---called *attribute function*---that maps occurrences in the occurrence domain Δ_f to attribute names in T . Then, for each occurrence $a \in \Delta_f$, we store sort symbol $\psi_f(a)$ in attribute $v(a)$ in tuple r .

This representation is sound, but it can be compressed by recognizing that for particular occurrences in the occurrence domain, symbols are the same in all facts in the segment. For example, the symbol at the *who* occurrence in Example 3.1 is *mary* for all facts in Q . This (possibly empty) set of occurrences is the *fixed symbol set*:

Definition 18 (Fixed symbol set) *For qualified segment Q we define the fixed symbol set $D_Q \subseteq \Delta_Q$ as:*

$$D_Q = \{a \in \Delta_Q \mid \forall f, f' \in Q : \psi_f(a) = \psi_{f'}(a)\}$$

⁵More precisely, this is true if the qualified segment is not reduced to only one fact. But then, as we shall see, there is no relation to store in the database.

Symbols at occurrences in the fixed occurrence set D_Q are the same for all facts in qualified segment Q , hence, we do not have to store them in the database. We only store symbols at occurrences not in D_Q and use any basic term in the qualifier to represent the missing symbols. Indeed, for each basic term q in the qualifier, the sort symbol $\psi_q(a)$ for each occurrence a in the fixed symbol set D_Q is their LUB and thus the same as the symbol at this occurrence for all facts in Q .

The correspondence between qualified segment Q and database relation R_T is defined by a *data definition*:

Definition 19 (Data definition) *Given segment Q , the corresponding database relation R_T is defined by a data definition given by the quadruple $\langle \mathbf{qua}(Q), R, v, D_Q \rangle$.*

Data definitions are stored in the interface, thus enabling the representation of facts in segment Q as tuples in R_T . With each fact $f = \langle \Delta_f, \psi_f \rangle \in Q$ corresponds a unique tuple $r \in R_T$, defined by:

$$\forall t \in T : r.t = \psi_f(v^{-1}(t))$$

Conversely, each database tuple $r \in R_T$ represents a fact $f = \langle \Delta_Q, \psi_f \rangle$, where the sort function ψ_f is defined as:

$$\psi_f(a) = \begin{cases} \top & \text{if } a \notin \Delta_Q \\ \psi_q(a) & \text{if } a \in D_Q \\ r.v(a) & \text{otherwise} \end{cases}$$

where $q \in \mathbf{qua}(Q)$.

Example 3.2 The qualifier for qualified segment Q from Example 3.1 is $\{\text{likes}(\text{who} \Rightarrow \text{mary}, \text{what} \Rightarrow \text{food})\}$, and the fixed symbol set is $D_Q = \{\varepsilon, \text{who}\}$. If we represent Q as a database relation R_T , we only need to store the symbols at occurrence *what*, so we need a relation with a single column, say $T = \{\text{foodname}\}$.

We define the attribute function v as: $v(\text{what}) = \text{foodname}$. The representation of Q as a database relation is $R_T = \{\langle \text{sweets} \rangle, \langle \text{apples} \rangle\}$. \square

Note, for the sake of consistency, that in the already mentioned degenerate case of a qualified segment reduced to only one fact, all the information goes into the fixed address set and the qualifier, leaving nothing to be stored in the external database.

4 Retrieval algorithm

For the retrieval of facts from the database, we use a tight coupling, where we load facts from the database whenever needed by the inference engine. For a particular goal g , we load the subset $Q[g]$ from segment Q , containing all facts in Q that unify with g :

$$Q[g] = \{f \in Q \mid f \wedge g \neq \perp\}$$

Qualified segment Q is stored in the database, so we do not know its actual contents, hence we cannot compute $Q[g]$ by simply unifying all facts in Q with the goal. So, we need another technique to compute $Q[g]$, independent of the contents of Q . We use an *abstract interpretation* [11] of the inference process, where we use qualifiers instead of facts. In this abstraction, unification of facts in Q with goal g is an operation on the qualifier and the goal, resulting in a term---called the *candidate*---which approximates the subset of Q of all facts unifiable with g . We describe the construction of candidates. First, we define the *unifiable set* $U(s)$, the set of all sort symbols that unify with symbol s ; *i.e.*, symbols for which the maximal common subsort with s is non-bottom:

Definition 20 (Unifiable set) For a sort symbol s in \mathcal{S} , we define the unifiable set $U(s)$ as:

$$U(s) = \{s' \in \mathcal{S} \mid s \sqcap s' \neq \{\perp\}\}$$

A candidate is defined such that any fact in the qualified segment subsumed by a basic term in the candidate, unifies with goal g :

Definition 21 (Candidate) Given a goal g , a basic term, the candidate C is the set of all maximal terms $c = \langle \Delta_Q, \psi_c \rangle$ that can be constructed from a term q in the qualifier $\mathbf{qua}(Q)$ that is unifiable with g , as follows. $\forall a \in \Delta_Q$:

$$\psi_c(a) \begin{cases} = \top & \text{if } a \notin D_Q, \text{ or } \psi_q(a) \leq \psi_g(a), \\ \in \mathbf{chi}(\psi_q(a)) \cap U(\psi_g(a)) & \text{otherwise.} \end{cases}$$

Example 4.1 Assume the goal $g_1 = \text{likes}(\text{what} \Rightarrow \text{cookies})$ and qualified segment Q as in Example 3.2. By Definition 21, we construct a candidate $C_1 = \top(\text{who} \Rightarrow \top, \text{what} \Rightarrow \text{sweets})$. For goal $g_2 = \text{likes}(\text{who} \Rightarrow \text{student}, \text{what} \Rightarrow \text{food})$, we construct candidate $C_2 = \top(\text{who} \Rightarrow \top, \text{what} \Rightarrow \top)$. For goal $g_3 = \text{likes}(\text{who} \Rightarrow \text{peter}, \text{what} \Rightarrow \text{apples})$, we construct candidate $C_3 = \emptyset$. \square

Thus a candidate contains terms, identically formed to the facts in the segment, and consisting of \top -symbols and immediate subsorts of symbols in the qualifier; *i.e.*, symbols that appear in facts in Q . If candidate C is empty, the symbols in the terms in the qualifier and the goal do not unify, then the qualified segment does not contain any facts that unify with the goal. We have to prove that any fact f in qualified segment Q that unifies with goal g , is subsumed by a basic term c in candidate C .

Theorem 2 A fact f in qualified segment Q unifies with goal g iff it is subsumed by a basic term c in candidate C ; namely,

$$f \wedge g \neq \perp \Leftrightarrow f \leq c$$

Proof By Definition 13 and Theorem 1, we can rewrite the above to a condition on sort symbols, $\forall a \in \mathcal{L}^*$:

$$\psi_f(a) \sqcap \psi_g(a) \neq \{\perp\} \Leftrightarrow \psi_f(a) \leq \psi_c(a)$$

We first prove that if the maximal common subsort of two symbols $\psi_f(a)$ and $\psi_g(a)$ is non-bottom, then we can construct a term c such that $\psi_f(a)$ is smaller than the corresponding symbol $\psi_c(a)$ in c .

Symbols $\psi_f(a)$ and $\psi_g(a)$ unify, so $\psi_f(a)$ is in the unifiable set $U(\psi_g(a))$. Symbol $\psi_q(a)$ is larger than $\psi_f(a)$, and thus unifies with $\psi_g(a)$ as well: $\psi_g(a) \in U(\psi_q(a))$. So, by definition, $\psi_c(a)$ is not the symbol \perp . Assume that occurrence a is in the fixed occurrence set D_Q . By definition, $\psi_c(a) = \top$ and thus symbol $\psi_f(a)$ is smaller than the symbol $\psi_c(a)$ in c . Alternatively, if occurrence a is not in the fixed symbol set D_Q , symbol $\psi_f(a)$ in fact f is a child of $\psi_q(a)$. We also know that $\psi_f(a)$ is in $U(\psi_g(a))$, thus we can construct a term c where $\psi_c(a) = \psi_f(a)$. So we can construct a term c larger than any fact f that unifies with goal g .

We also prove that if fact f in Q does *not* unify with goal g , we cannot construct a term c larger than f . Fact f and term g do not unify, so for at least one occurrence a , the maximal common subsort of $\psi_f(a)$ and $\psi_g(a)$ is the bottom symbol. We prove that, for this occurrence, we cannot construct a candidate c with $\psi_f(a) \leq \psi_c(a)$.

The symbol $\psi_q(a)$ is a supersort of $\psi_f(a)$. If q and g do not unify, the candidate is empty. Thus, it does not subsume any fact. If q and g unify then $\psi_q(a)$ is in $U(\psi_g(a))$, for all occurrence a in Δ_Q . Symbol $\psi_g(a)$ cannot be a supersort of $\psi_q(a)$, otherwise, $\psi_g(a)$ would be a supersort of $\psi_f(a)$ as well, and their maximal common subsort would be $\psi_f(a)$. Moreover, occurrence a cannot be in the fixed symbol set D_Q , otherwise $\psi_f(a) = \psi_q(a)$, contradicting that $\psi_q(a)$ is not in the unifiable set $U(\psi_g(a))$. Hence, the symbol $\psi_c(a)$ in c is not \top .

If we can construct a term c larger than f , symbol $\psi_c(a)$ would be a child of $\psi_q(a)$ and a member of the unifiable set $U(\psi_g(a))$. Since occurrence a is not in the fixed occurrence set, $\psi_f(a)$ is also a child of $\psi_q(a)$. So the only child of $\psi_q(a)$, larger than $\psi_f(a)$, is $\psi_f(a)$ itself. However, $\psi_f(a)$ is not in the unifiable set $U(\psi_g(a))$, so we cannot construct a term c , where $\psi_c(a) \in \mathbf{chi}(\psi_q(a)) \cap U(\psi_g(a))$, that is larger than fact f . ■

Corollary 1 *If fact f is subsumed by a basic term c in candidate C , all symbols in c are either the top symbol, or equal to the corresponding symbol in fact f .*

Proof Follows directly from the above proof, since $\psi_c(a)$ is either \top , or a child of the symbol $\psi_q(a)$ in the qualifier. For these symbols, occurrence a is not in the fixed occurrence set, thus symbol $\psi_f(a)$ in term f is also a child of $\psi_q(a)$. ■

The corollary is important, since it states that we can compute $Q[g]$ by a selection with the candidates, where \top is the wild card argument and non-top symbols are selection arguments. With a candidate C for data definition $D = \langle F, R_T, v, D_Q \rangle$, there corresponds a selection condition $T[C]$ that is true for all elements of the set $Q[g]$ and false for any other element of Q :

$$T[C] = (T[c_1]) \mathbf{or} \dots \mathbf{or} (T[c_p])$$

where $C = \{c_1, \dots, c_p\}$. For each term c_i we construct a selection condition:

$$T[c_i] = \begin{array}{l} (v(a_1) = \psi_c(a_1)) \\ \mathbf{and} \quad \dots \\ \mathbf{and} \quad (v(a_n) = \psi_c(a_n)) \end{array}$$

where a_1, \dots, a_n are the occurrences with non-top symbols in term c_i . We select the tuples that represent facts in $Q[g]$ with a simple SQL-query:

```

select   $t_1, \dots, t_n$ 
from     $R$ 
where    $T[C]$ 

```

The retrieved tuples are then translated to facts, as stated in Section 3.2.

Example 4.2 For the candidate C_1 of Example 4.1, we construct a selection condition $T[C_1] = (v(\mathit{what}) = \psi_{c_1}(\mathit{what})) = (\mathit{foodname} = \mathit{sweets})$. The query is:

```

select   $\mathit{foodname}$ 
from     $R$ 
where    $\mathit{foodname} = \mathit{'sweets'}$ 

```

and returns the tuple $\langle \mathit{sweets} \rangle$, which is transformed to the fact $\mathit{likes}(\mathit{who} \Rightarrow \mathit{mary}, \mathit{what} \Rightarrow \mathit{sweets})$. \square

5 Reduced sort signature

For the construction of candidates, we use sort signature Σ . Part of the subsort relationships are implicitly represented in the database, that is, for each fact in a qualified segment, the parents of all symbols at occurrences not in the fixed symbol set D_Q are stored in the qualifier. We do not store these ‘implicit’ subsort relationships in the LIFE system, but add them when facts are loaded.

The remaining subsort relationships have to be stored in the LIFE system, since we have to be able to reconstruct the entire sort signature. However, part of the subsort relationships implicitly stored in the database are needed to construct candidates. Thus we should either retrieve these relationships at run-time from the database, or simply duplicate the necessary relationships in the LIFE system, or use a combination of both techniques.

We will adopt the second strategy, which is simple, and probably non-optimal: we store sufficient subsort relationships in the LIFE system to compute candidates for any goal and qualifier in program P . We construct a *reduced sort signature* $\Sigma' = \langle \mathcal{S}', \leq' \rangle$, where $\mathcal{S}' \subseteq \mathcal{S}$ and $\leq' \subseteq \leq$.

Definition 22 (Reduced sort signature) *Given $\Sigma = \langle \mathcal{S}, \leq \rangle$, the reduced sort signature $\Sigma' = \langle \mathcal{S}', \leq' \rangle$ is such that \mathcal{S}' is the subset of \mathcal{S} , where we may exclude least sorts (parents of bottom) with a single parent, stored in a database relation, and not in a term in a qualifier. The reduced subsort relation \leq' is the subset of \leq , induced by the set \mathcal{S}' :*

$$\leq' = \leq \cap \mathcal{S}' \times \mathcal{S}'.$$

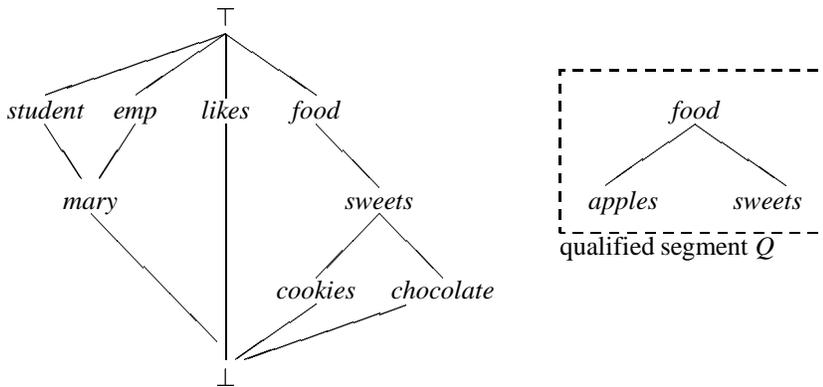


Figure 3. Reduced sort signature Σ' .

Example 5.1 The reduced sort signature Σ' is depicted in Figure 3. The least sorts with a single parent are the symbols *likes*, *apples*, *cookies* and *chocolate*. The symbols in the database are *apples* and *sweets*. The symbols not in a qualifier are *student*, *emp*, *apples*, *sweets*, *cookies* and *chocolate*. Hence, the only symbol that is a least sort, in a database relation and not in a qualifier is *apples*. \square

We have to prove that the reduced sort signature is complete; that is, all subsort relationships are represented either in the database or in the reduced sort signature. Moreover, we have to prove that we construct the same candidates with the reduced sort signature.

Theorem 3 *All subsort relationships are either represented in the LIFE system or implicitly in the database.*

Proof Assume a subsort relation $s \leq s'$ where s is not in \mathcal{S}' . By definition, s is a symbol in a database relation, and not a symbol in a qualifier. So there is a symbol $s'' \in \mathcal{S}'$ at the corresponding occurrence in the qualifier for this database relation, so $s \leq s''$ is a relation implied by this segment. Since s and s'' are in \mathcal{S}' , $s'' \leq' s'$. So we can reconstruct $s \leq s'$, since $s \leq s''$ and $s'' \leq' s'$.

Now assume the relation $s \leq s'$ where s' is not in \mathcal{S}' . Since only least sorts are not stored in \mathcal{S}' , s must be the bottom symbol, and $\perp \leq s'$ is implicitly defined by the sort signature for any $s' \in \mathcal{S}$. \blacksquare

Theorem 4 *If we exchange Σ for Σ' , we construct the same candidates for a goal g and a qualifier $\text{qua}(Q)$.*

Proof To construct candidates, we compute the unifiable set $U(s)$ for any symbol s in the goal. We define $U'(s)$ as the set containing all symbols in \mathcal{S}' that unify with $s \in \mathcal{S}'$, as defined by the subsort relation \leq' . For the correct construction of candidates, $U'(s)$ should contain all symbols in $U(s)$ that are also in \mathcal{S}' , that is:

$$\forall s, s' \in \mathcal{S}' : s' \in U'(s) \Leftrightarrow s' \in U(s)$$

Symbol s' is in $U(s)$ if the maximal common subsort of s and s' is non-bottom. We prove that for any s, s' in \mathcal{S}' , maximal common subsorts $s \sqcap s'$ form a subset of \mathcal{S}' , and thus that s' is in $U'(s)$ if s' is in $U(s)$. The set $s \sqcap s'$ is either $\{s\}$ or $\{s'\}$, or a set of symbols, smaller than both s and s' . These symbols are all in \mathcal{S}' , since we excluded only symbols with a single parent, thus symbols that can never be a maximal common subsort of two other symbols.

Moreover, if $s \sqcap s' = \{\perp\}$ (i.e., $s' \notin U(s)$), then s' is not in the unifiable set $U'(s)$ as well, since the subsort relation \leq' in the reduced sort signature form a subset of the subsort relation \leq . ■

As can be seen in Example 5.1 and Figure 3, simply duplicating all necessary subsorting information works fine for qualified segments containing a large number of facts with least sort symbols (i.e., data typically found in databases), since these symbols are not stored in the reduced sort signature. However, we stress that the above solution is non-optimal, since the reduced sort signature Σ' contains more subsort information than actually needed. We believe it is possible to further ‘strip-down’ the reduced sort signature. We think of a technique called *segment guessing*, where less subsort information is needed, and the retrieval algorithm queries any database relation that might contain unifiable facts, based on available subsort information.

6 Optimization

To reduce database interaction, we assert loaded facts in the internal LIFE database, instead of retrieving the same facts over and over again. However, if we assert facts in the internal database, we should retrieve each fact only once. Thus when querying the database for all unifiable facts for goal g_i in segment Q , we should exclude all facts loaded from Q for previous goals g_1, \dots, g_{i-1} .

As we stated in Section 4, we can describe each subset $Q[g_i]$ with a selection condition $T[C_i]$. Thus we can exclude any subset with the negation of its selection condition. We select the tuples from the database with an SQL-query:

```

select   $t_1, \dots, t_n$ 
from     $R$ 
where    $T[C_i]$  and not ( $T[C_1]$ )
           and    ...
           and not ( $T[C_{i-1}]$ )

```

The set of all candidates for previous goals forms an *abstract cache*, storing the results of previous abstract computations; i.e., all constructed candidates. This is also known as the *caching of queries*, as described by Ceri *et al.* in [10]. However, storing all these candidates is expensive, and therefore we will shortly mention a few optimizations.

Instead of storing all previous candidates, we use a single set---called *look-up set* to represent that part of the qualified segment that has been loaded:

Definition 23 (Look-up set) For a segment Q , we define the look-up set $L[i]$ as the set, formed of the maximal terms in the union of candidates c_1, \dots, c_i .

A look-up set is an equivalent, but more compact notation for a set of candidates, since any term subsumed by another term, is removed. The SQL-query reduces to:

```

select   $t_1, \dots, t_n$ 
from     $R$ 
where    $T[C_i]$  and not ( $L[i - 1]$ )

```

Another optimization consists of posing only queries that might retrieve any tuples, that is, we exclude queries with a contradicting selection condition. This occurs when the current query is subsumed by a previous query, as described in [10]. The subsumption of queries is defined by the subsort relation \leq on candidates. That is, all facts for goal g_i have been loaded if any term c in candidate C_i is subsumed by some term c' in the look-up set: $\forall c \in C_i, \exists c' \in L[i - 1] : c \leq c'$.

A third optimization is the partial exclusion of previous queries. If we retrieve a set from the database, we only need to exclude previously retrieved sets that overlap with the current set; *i.e.*, $Q[g_i] \cap Q[g_j] \neq \emptyset$.

We further like to mention that, since candidates are wild card selections, testing subsumption and overlapping reduces to simple comparison operations on the respective sort symbols.

7 Conclusion

We have overviewed a formal design for interfacing a logical query language with complex objects to a relational database. Our system is an improvement on previous systems in that it provides database storage for objects ordered thanks to a subsort hierarchy, representing part of this hierarchy in the database as well. The representation of the objects is flexible; arbitrarily nested objects can be represented in a maximally compressed format, where compressing and decompressing is handled by the interface. The loading algorithm is quite efficient in that it loads only objects actually needed by the LIFE system, and never loads the same object twice, thus improving results in [10]. In addition, our design also improves on previous work by providing for free the ability, intrinsic to ψ -terms, to store and query partial information. For example, if all facts in LIFE's EDB stipulate that all students are happy, a query requesting to list happy things will avoid itemizing *in extenso* all 12,452 tuples of students, giving only the one tuple corresponding to the *intensional* LIFE fact $happy(student)$.

LIFE is an extension of logic programming: first-order logic programs are LIFE programs with a *flat* sort signature; *i.e.*, all sort symbols---except for \top and \perp are incomparable. Hence, the retrieval algorithm holds for languages using Prolog terms as objects as well.

Part of the system described in this paper has been implemented: the LIFE-WISDOM system (*LIFE With Inheritance Supported Data Object Management*) implements a database interface for an implementation of LIFE called WildLIFE [3], to an ORACLE relational database [12]. The current system

implements both database retrieval and updates, but only for single inheritance and facts consisting of least sorts.

As for the future, we want to extend this approach to goals with variables. For example, a goal such as $name(X, X)$ must only unify with facts with identical arguments and should generate database queries retrieving only tuples with identical values in columns. Then, we may translate entire LIFE rules to complex join operations on the database. The translation of recursive LIFE rules to extended relational algebra expressions must also be explored. Another direction of research consists of weakening the restrictions for the reduced sort signature, by redefining qualified segments and using other search strategies, such as *segment guessing*. Also, we may consider iterating our construction, building multiple levels of abstractions; *i.e.*, the storage of qualifiers themselves in *higher-level* qualified segments.

Acknowledgements

The authors wish to thank Herman Balsters, Jean-Pic Berry, Maurice van Keulen, and Andreas Podelski for their support and useful remarks.

References

1. Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293--351 (1986).
2. Hassan Aït-Kaci. An introduction to LIFE--programming with logic, inheritance, functions, and equations. In Dale Miller, editor, *Proceedings of the International Symposium on Logic Programming (Vancouver, BC)*, pages 52--68, Cambridge, MA (October 1993). MIT Press.
3. Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, and Peter Van Roy. The Wild_LIFE handbook. PRL Research Report (forthcoming), Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1994).
4. Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185--215 (1986).
5. Hassan Aït-Kaci, Roger Nasr, and Jungyun Seo. Implementing a knowledge-based library information system with typed Horn logic. *Information Processing & Management*, 26(2):249--268 (1990).
6. Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3-4):195--234 (July-August 1993).
7. François Bancilhon and Setrag Khoshafian. A calculus for complex objects. *Journal of Computer and System Sciences*, 38(2):326--340 (April 1989).
8. O. Peter Buneman, Susan D. Davidson, and Aaron Watters. A semantics for complex objects and approximate answers. *Journal of Computer and System Sciences*, 43(1):170--218 (August 1991).
9. Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer Verlag, Berlin, Germany (1990).

10. Stefano Ceri, Georg Gottlob, and Gio Wiederhold. Interfacing relational databases and Prolog efficiently. In Larry Kerschberg, editor, *Proceedings of the 2nd International Conference on Expert Database Systems*, pages 141--153, Menlo Park, CA (1987). Benjamin-Cummings.
11. Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103--179 (1992).
12. Marcel Holsheimer. LIFE--WISDOM, a database interface for the LIFE system. Master's thesis, Computer Science, University of Twente, Enschede, The Netherlands (September 1992).
13. Katherine Morris, Jeffrey D. Ullman, and Allen Van Gelder. Design overview of the Nail! system. In Ehud Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, pages 544--568, Berlin, Germany (1986). LNCS 225, Springer-Verlag.
14. Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, Rockville, MD (1989).
15. Yannis Vassiliou, James Clifford, and Matthias Jarke. How does an expert system get its data? In *Proceedings of the International Conference on Very Large Databases*, pages 70--72 (1983). Extended abstract.
16. Yannis Vassiliou and Matthias Jarke. Databases and expert systems: Opportunities and architectures for integration. In *New Applications of Databases*, pages 185--201, London, UK (1984). Academic Press.