Implementing a Knowledge-Based Library Information System with Typed Horn Logic[1]

Hassan Aït-Kaci
Roger Nasr

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, TX 78759

Jungyun Seo

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

Abstract

This article discusses the implementation of a knowledge base for a library information system. The system was conceived using a typed logic programming language—LOGIN—where type inheritance is built in. The knowledge base is structured in a hierarchical taxonomy of library object classes where each class is represented in a FRAME style knowledge structure and inherits the properties of its parents, and where infrastructural inference rules have been established through typed Horn clauses. Also in this document, some programming techniques aimed at using the power of inheritance as taxonomic inference are discussed.

[1]This article is a revised and extended version of [Aït-Kaci 88].

[...] solution to the ancient problem: *The library is unlimited and cyclical.* If an eternal traveler were to cross it in any direction, after centuries he would see that the same volumes were repeated in the same disorder (which, thus repeated, would be an order: the Order).

JORGE LUIS BORGES, *The Library of Babel.*

The practical upshot of all this is that if you stick a Babel fish in your ear you can instantly understand everything said to you in any form of language.

DOUGLAS ADAMS, *The Hitchhicker's Guide to the Galaxy.*

1 Introduction

The idea of attribute inheritance has been adopted by various programming languages. In particular, so called object-oriented programming languages have shown that this idea can be very practical in solving some knowledge representation problems. [Goldberg 80] [Stefik 83]

Aït-Kaci and Nasr [Aït-Kaci 86b] have proposed a new programming language, LOGIN, combining the idea of inheritance with logic programming. LOGIN replaces Prolog's first-order terms with ψ-terms which generalize first-order terms by allowing partially ordered constructors. LOGIN uses the standard Prolog operational semantics—a computation mechanism which implements natural deduction—with a ψ-term unification algorithm rather than an ordinary unification algorithm. It becomes natural to express FRAME style knowledge structures [Bobrow 77][Minsky 75] using ψ-terms in LOGIN. Furthermore, the strategies of ψ-term unification provides efficient expressions of set theoretical operations.

This paper describes an experimental library system called BABEL as a *concrete* application written in LOGIN.[2] The motivation for this work is the need for testing the design concepts of LOGIN, specifically showing its usefulness as a knowledge and database language. We believe that our learning experience from the design and implementation of practical applications is a necessary complement to our theoretical research.

In its current state of design, BABEL consists of four parts: the *transaction manager*, the *query manager*, the *natural language interface*, and the *knowledge base.* The transaction manager keeps track of the librarian's traditional duties: checking books in and out, reminding borrowers of overdue materials, assessing fines, and reserving loaned books for the next user. The query manager provides the library's users with information normally found in card catalogs. In BABEL, the query manager is an interactive query generator. The query manager accepts user queries through menus and generates the equivalent LOGIN queries. The natural language interface accepts natural language queries and generates equivalent LOGIN queries.[3] The knowledge base in BABEL consists of two parts. One is static and a representation of the library's *structural* information; the other is dynamic, a representation of its *assertional* information. The structural representation is formed by organizing library materials into a hierarchy of object classes where each class is given a (type) definition. A set of such definitions constitutes a formal *knowledge base* as defined in [Aït-Kaci 86a]. The *assertional* information is expressed in an order-sorted typed Horn logic [Smolka 87], whose types are drawn from the knowledge base. Such logical sentences, in the form of rules and facts (typed Horn clauses), are used to

- maintain time-dependent records—*e.g.*, items on loan, on reserve, on shelf, being recalled, *etc.*

[2]**B**ABEL: **A**uthentique **B**ibliothèque **É**crite en **L**OGIN.

[3]The natural language query interface manager is not included in this report and will be implemented in the future.

- express queries—*e.g.*, itemize all library users holding an overdue item.

The library's *extensional* database (*i.e.*, the object values) consists of the individual library items and library users.

Although we introduce LOGIN briefly in Section 2, we shall assume for the most part that the reader is familiar with logic programming. In Section 3, a quick point is made to illustrate the LOGIN way of representing some BABEL library object as opposed to the Prolog way. In Section 4, our method of building a knowledge base for library information using a hierarchical classification of library objects is presented. The transaction manager and the query manager are described in Section 5. Finally, a brief conclusion speculating on this experiment is drawn in Section 6.

2 LOGIN: an Overview

LOGIN [Aït-Kaci 86b] is an elaboration of Prolog. The main extension is its new definition of terms which are arguments of literals. In first-order logic, a *literal* is 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. LOGIN extends the first-order terms of Prolog into partially ordered (first-order) type structures called ψ-terms. A ψ-term denotes a class of objects. Unification of ψ-terms denotes class intersection. The least specific class (the universe) is the largest type ($\top$), and the over-specified class (the empty set) is the smallest (uninhabited) type ($\bot$).

Informally, a ψ-term consists of:

1. A *root symbol* which is a type constructor and denotes a class of objects.
2. *Attributes*, label and value pairs, which are record fields. Each label is associated with a sub-ψ-term as a value for the label. The sub-ψ-term can be a constructor, a typed variable called tag, or a ψ-term.

An example of a ψ-term is:

```
book(title => string,
     author => name(fname => X: string,
                    lname => X),
     call_number => lc_number,
     pub_date => date(day => integer,
                      month => monthname,
                      year => integer),
     isbn => string)
```

The root symbol is *book*—a type constructor—which shows the class that this (ψ-term) type expression denotes. It has five sub-ψ-terms under the attribute labels *title*, *author*, *call_number*, *pub_date*, and *isbn*. This ψ-term denotes a class of book whose authors have the same first and last names, *e.g.*, Allen Allen.

Each class inherits the attributes of its parents in the class hierarchy—*i.e.*, if a class **c** has a subclass **d**, **d** inherits all of **c**'s attributes. For example, we can define an ordering between the two type symbols (denoting classes) *library_material* and *book* as follows. Given a definition of *library_material*[4] with its attributes and their types (no root symbol means $\top$),

[4]This definition of *library_material* can be read as '*library_material* is a set of objects of any type with four attributes...'

```
library_material = (title => string,
                    author => name,
                    call_number => lc_number,
                    pub_date => date).
name = (fname => string,
        lname => string).
date = (day => integer,
        month => monthname,
        year => integer).
```

we can define *book* as:

```
book = library_material(isbn => string).
```

which is the same as:

```
book = library_material(title => string,
                        author => name,
                        call_number => lc_number,
                        pub_date => date,
                        isbn => string).
```

We call *book* a *subtype* (subclass) of *library_material*. This definition of *book* can be read as "*book* is a subset of *library_material* and has one more attribute—ISBN (International Standard Book Number)".

Given a partially ordered type hierarchy, we can implement a generalized unification algorithm between ψ-terms, interpreted as set intersection. The following illustrates the ψ-term unification algorithm [Aït-Kaci 86b]. Given a type hierarchy in Figure 1, unifying the ψ-term:

```
 book(title => X,
      author => (lname => 'Winston'),
      call_number => Y)
```

and the ψ-term:

```
ai_material(title => Z,
            author => (fname => W),
            pub_date => (year => 1985))
```

results in the ψ-term:

```
 ai_book(title => X,
         author => (fname => W,
                    lname => 'Winston'),
         call_number => Y,
         pub_date => (year => 1985))
```

Unifying these two ψ-terms—*book* and *ai_material*—become *ai_book*. This is precisely the intersection of two sets, the set of all AI material and the set of all books (in the library), which results in the set of all AI books. As illustrated in above example, the set of the attributes in a resulting ψ-term from the unification of two ψ-terms is the same as the *union* of attributes in the two ψ-terms. Taking a *union* of attributes imposes more restrictions on the resulting ψ-term. Restricting a ψ-term by assigning constraints (values) to attributes thus specifies a subset of a set. For example, the ψ-term

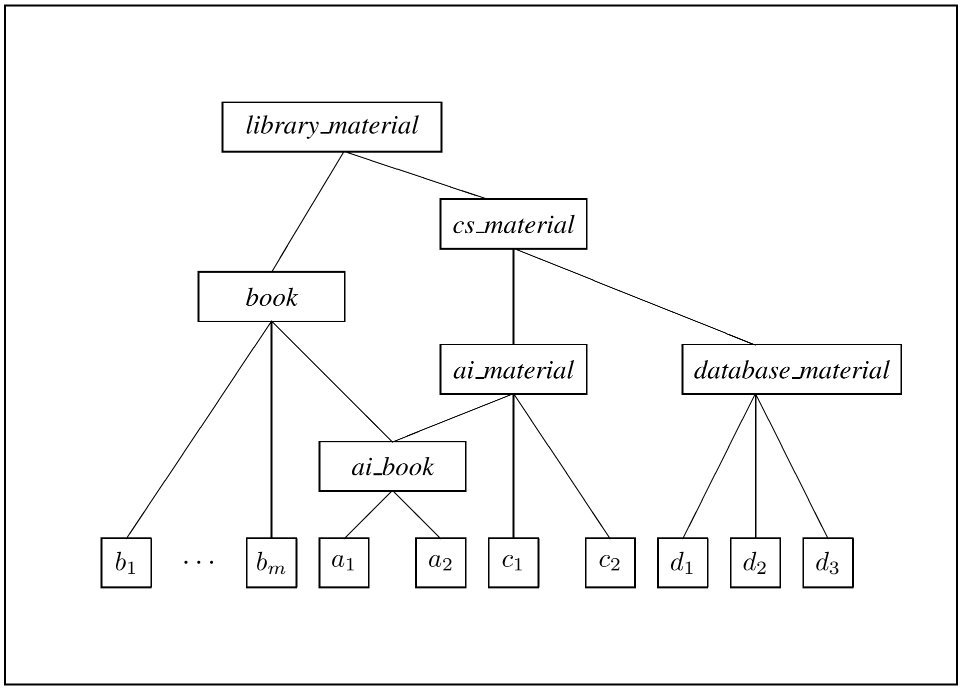

Figure 1: A Type Hierarchy with Multiple Inheritance

```
book(author => (lname => 'Winston'))
```

denotes books written by Winston, a subset of all books.

Informally,[5] unification of two ψ-terms fails if the root symbols—-type constructors— are not unifiable in partially ordered type hierarchy, or the two ψ-terms have at least one same attribute label but with values which are not ψ-term unifiable. Otherwise, the unification succeeds.

In LOGIN, one can explicitly define a disjunctive class, which is not allowed in other frame representation languages [Bobrow 77][Goldstein 77]. For example, we can redefine a ψ-term `date` as

```
date = (day => integer,
        month => {integer ; monthname},
        year => integer).
monthname = {'Jan'; 'Feb'; 'Mar'; 'Apr'; 'May'; 'Jun';
             'Jul'; 'Aug'; 'Sep'; 'Oct'; 'Nov'; 'Dec'}.
```

It means that the attribute `month` can have a value of type `integer` or `monthname`, and any of the twelve character strings can be a valid value of type `monthname`.

Again, LOGIN is simply Prolog where first-order terms are replaced by ψ-terms. Thus, LOGIN uses the standard Prolog operational semantics (SLD-resolution) with a ψ-term unification algorithm rather than an ordinary unification algorithm (order-sorted SLD-resolution [Smolka 87].)

[5] See [Aït-Kaci 86b] for rigorous definitions and algorithms.

Since ordinary first-order terms in Prolog can be represented as ψ-terms, LOGIN is more expressive than Prolog, and subsumes it. For example, an ordinary term in Prolog $f(a, g(b, X))$ is equivalently represented in LOGIN as the ψ-term:

```
f(1 => a,
  2 => g(1 => b,
         2 => X))
```

where the function symbols, f and g, in ordinary terms become type symbols in ψ-terms.

Unlike Prolog, the arity and the order of arguments (attributes) in ψ-terms are *transparent* to the user, in the sense that one may specify only subsets of them, and in any order.[6]

3 Knowledge Representation in LOGIN

Let us consider an example. A record for a book in the library database such as

```
Title       :  The Handbook of Artificial Intelligence
Author      :  Avron Barr, Edward A. Feigenbaum
Call Number :  QA76.65 b77
Subject     :  Artificial Intelligence
```

can be represented in Prolog as, say,

```
library_item(type(book),
             title('The hand book of artificial intelligence'),
             author(['Avron Barr', 'Edward A. Feigenbaum']),
             call_number('QA76.65 b77'),
             subject(artificial_intelligence)).
```

Given this data, a user query such as:

"Show computer science books written by Edward A. Feigenbaum"

which would be expressed formally in Prolog as

```
query
:- library_item(type(book),
                title(X),
                author(Author_list),
                _,
                subject(computer_science)),
   member('Edward A. Feigenbaum',
          Author_list),
   show(X).
```

[6]As a syntactic sugar, LOGIN also supports positional terms in Prolog; *i.e.*, terms such as $f(a, g(b), X)$ and $f(a, g(b), num \Rightarrow X)$ are all valid ψ-terms. Arguments which have no labels are sensitive to the order.

would not succeed. The query fails because *subject(artificial_intelligence)* cannot be unified with *subject(computer_science)* in Prolog. In an intelligent library information system, a simple deduction like *"since Artificial Intelligence (AI) is one area of Computer Science (CS), all items in the AI area would be regarded as items in the CS area,"* is taken as an inference step.

In LOGIN, the same data may be represented as

```
artificial_intelligence < computer_science.
                        /* define hierarchy between two terms */

 b101 = book(title => 'The Handbook of Artificial Intelligence',
             author => { (fname => 'Avron',
                          lname => 'Barr') ;
                         (fname => 'Edward',
                          mname => 'A.',
                          lname => 'Feigenbaum')},
             call_number => 'QA76.65 b77',
             subject => artificial_intelligence).

library_item(b101).
```

So, the query:

```
query
:- library_item(book(author => (lname => 'Feigenbaum'),
                     subject => computer_science,
                     title => X)),
   show(X).
```

will succeed with $X =$*'The Handbook of Artificial Intelligence'*. Since *artificial_intelligence* is declared as a subtype of *computer_science*, *artificial_intelligence* can be ψ-term unifiable with *computer_science*. Furthermore, the disjunctive definition for the value of the attribute *author* make it possible to succeeds to get an answer in unification phase, not using inference step. Note that even though the query carries only the last name of the author, it succeeds to get an answer. This makes it possible to handle an incomplete query.

Note that number and order of arguments are transparent to the unification of two ψ-terms. Also, the reader may wonder why *library_item* is an assertional predicate rather that a structural type in the knowledge base like *library_material*. The reason is that the latter records static immutable information pertaining to the structure of a library object, while the former is needed as a handle to access the actual library records. Representation in current BABEL of a record's dynamic states (*e.g.*, on loan, on shelf, or current borrower, date due, *etc.*) will be discussed in Section 4.2.

Naturally, the problem can be solved in Prolog, but it requires more rules and facts in the inference system, such as:

```
is_a(X, _)
:- var(X),      /* to prevent infinite loop */
   fail, !.

is_a(X, X).
is_a(X, Y)
:- is_a(X, Z),
   is_a(Z, Y).

is_a(artificial_intelligence, computer_science).
```

We contend that, although semantically equivalent, the solution in Prolog is pragmatically inferior to that of LOGIN for the following reasons. First, in Prolog, the number and the order of arguments are not transparent to the programmer, noticeably losing perspicuity. Second, it is more desirable to express information into the type language (the partially-ordered knowledge base), whereby realizing a more restricted logic—albeit more efficiently—leaving to the more general and consequently less efficient deduction system only the tasks which require its differential power. In addition, this saves precious backtracking steps (this will be discussed in Section 4.2.)

4 Knowledge Base of BABEL

We now discuss the implementation of BABEL's knowledge base. The ordering structure of the knowledge base is set inclusion (*is-a*) between object classes. Each class is comprised of objects sharing certain characteristics. This is the representation of *structural* library information.

For example the class denoted by *book* is the class of all books. The type *cs_material* is a class of all materials in computer science. Items in *cs_material* can be books, theses, technical reports, microfilms, periodicals, or proceedings in computer science.

We can easily come up with a new class by combining two classes to make a more restricted class. For example, we can define a new object class, *cs_book*, which is the class of all computer science books by taking an intersection of the two classes *book* and *cs_material*. We call this new class a *subclass* of both *book* and *cs_material*—indeed, the *greatest* such class. At the same time, these two classes are called *super* classes of *cs_book*.

Each class may have arbitrarily many attributes represented as label-value pairs. A value can be a specific ground value or a typed/untyped variable. These attributes are inherited by lower classes.

For example, the class *cs_material* has an attribute *subject* with the value *computer_science*. Since *cs_book* is a subclass of *cs_material*, the attribute *subject* and the value *computer_science* are inherited by *cs_book*. Since the inheritance operation is type unification and denotes set intersection, the value of an attribute in a subclass takes priority over an inherited value of the same attribute.

In addition to *object* class hierarchies, BABEL maintains several *conceptual* term hierarchies in its knowledge base. Operationally and semantically there is no difference between the two, but the purpose of a conceptual hierarchy is different from that of an object hierarchy.

For example, *cs_book* has an attribute with label *subject* and value *computer_science*. The class *ai_book*, a subclass of *cs_book*, also has the same attribute, *subject*, but has a different value, *artificial_intelligence*. Since *ai_book* is a subtype of *cs_book*, *artificial_intelligence* should be a subtype of *computer_science*. Therefore, *artificial_intellig-ence* is classified as a subclass of *computer_science* in the conceptual term hierarchy of *subjects*. It does not describe a class of physical objects but rather a conceptual relationship between subjects.

4.1 Classes in the Knowledge Base

In this section, we examine the relationships among the different classes in the knowledge base.

The highest class in BABEL's physical library object taxonomy, *library_material*, encompasses all the library's contents. Each item in the library is then classified by its physical description—book, magazine, film, *etc.*—and also by subject—philosophy, social science, pure science, *etc.*

4.1.1 Classification by Physical Description

According to [Gorman 78], one possible taxonomy of library material by physical description is as follows:

- *printed_monograph*: printed materials including books, pamphlets, articles, and printed sheets. Subclasses include *monograph* and *pamphlet*.
- *cartographic_material*: geographical representations. Subclasses include *maps_atlases* and *globes*.
- *music_score*: subclasses include *vocal_score* and *instrument_score*.
- *sound_recording*: subclasses include *disc* and *tape*.
- *motion_picture*: subclasses include *motion_film* and *video_tape*.
- *micro_forms*: sublasses include *microfilm* and *microfiche*.

The above six subclasses of *library_material* are physically distinct. However, *serials*, another subclass of *library_material*, is not; in the sense that *serials* is a class of publications of *any type of medium* issued in succession, either numerically or chronologically, and intended to be continued indefinitely. We can generate new subclasses by combining other classes with the class *serials*. For example, the subclass *series_book* is a subclass of two classes, *series* and *book*. A part of the type hierarchy of library materials classified by physical description is in Figure 2).

4.1.2 Classification by Subject

This section describes BABEL's classification by subject. In this taxonomy, library materials are classified according to their subjects, regardless of physical description. Therefore, new subclasses can be generated by combining any classes in this scheme with any classes defined by physical description.

According to Dewey classification, all library items are divided into ten subject classes—independent of their physical description. A part of the type hierarchy of library materials classified by subject classes is illustrated in Figure 3. A taxonomy of library material by subject is obtained by further constraining the type of the *subject* attribute of *library_material* to be the appropriate specific subtype of *subjects*, from the conceptual subject hierarchy. Thus, such a type definition as:

```
philosophy_material
= library_material(subject => philosophy).
```

is made for all such material. In addition to *philosophy*, they include *applied_science*, *art*, *history*, *language*, *literature*, *pure_science*, *religion*, and *social_science*. For each of these—call it *???*—there is a corresponding type definition:

```
???_material
= library_material(subject => ???).
```

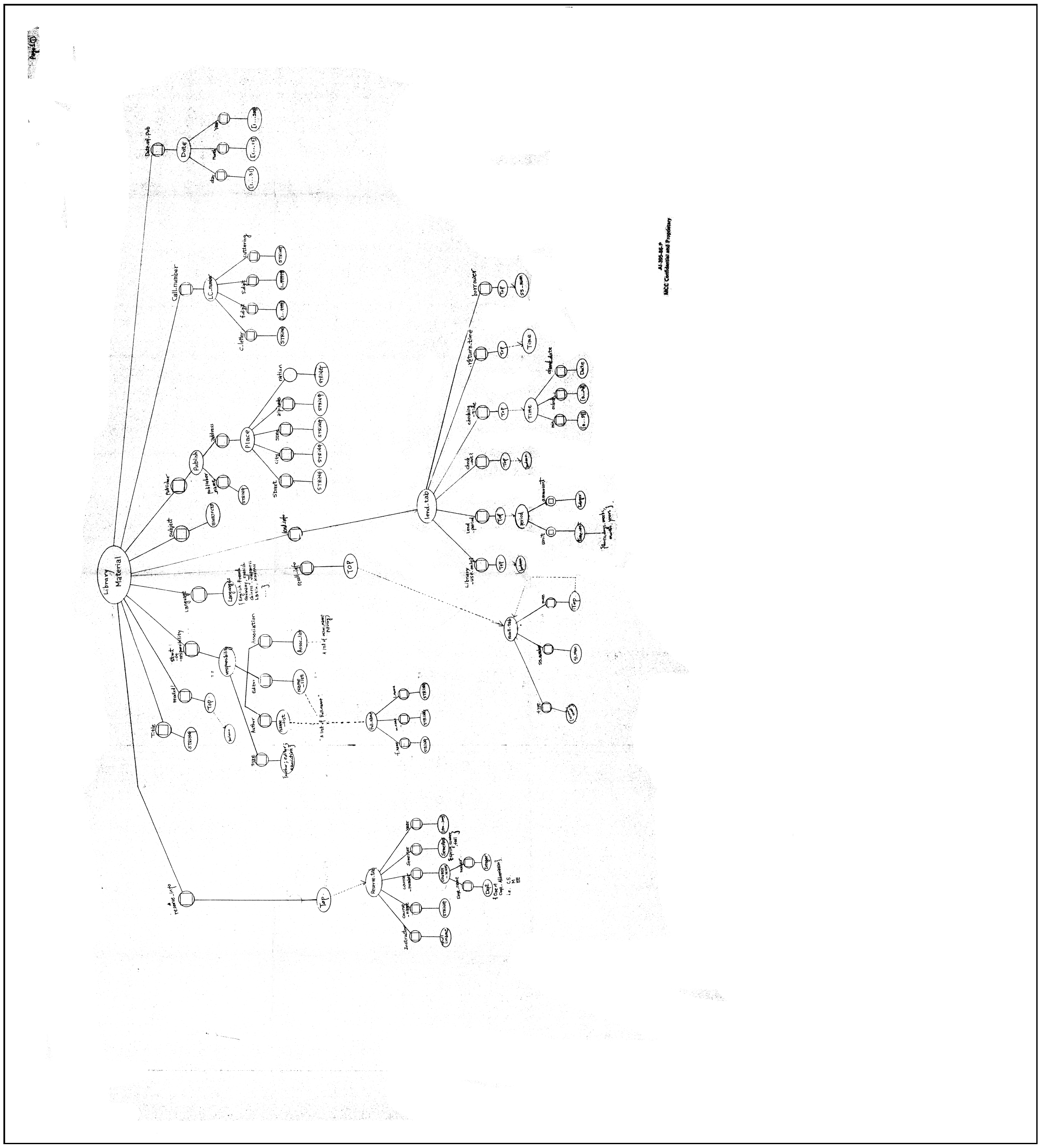

Figure 2: Hierarchy by Physical Description

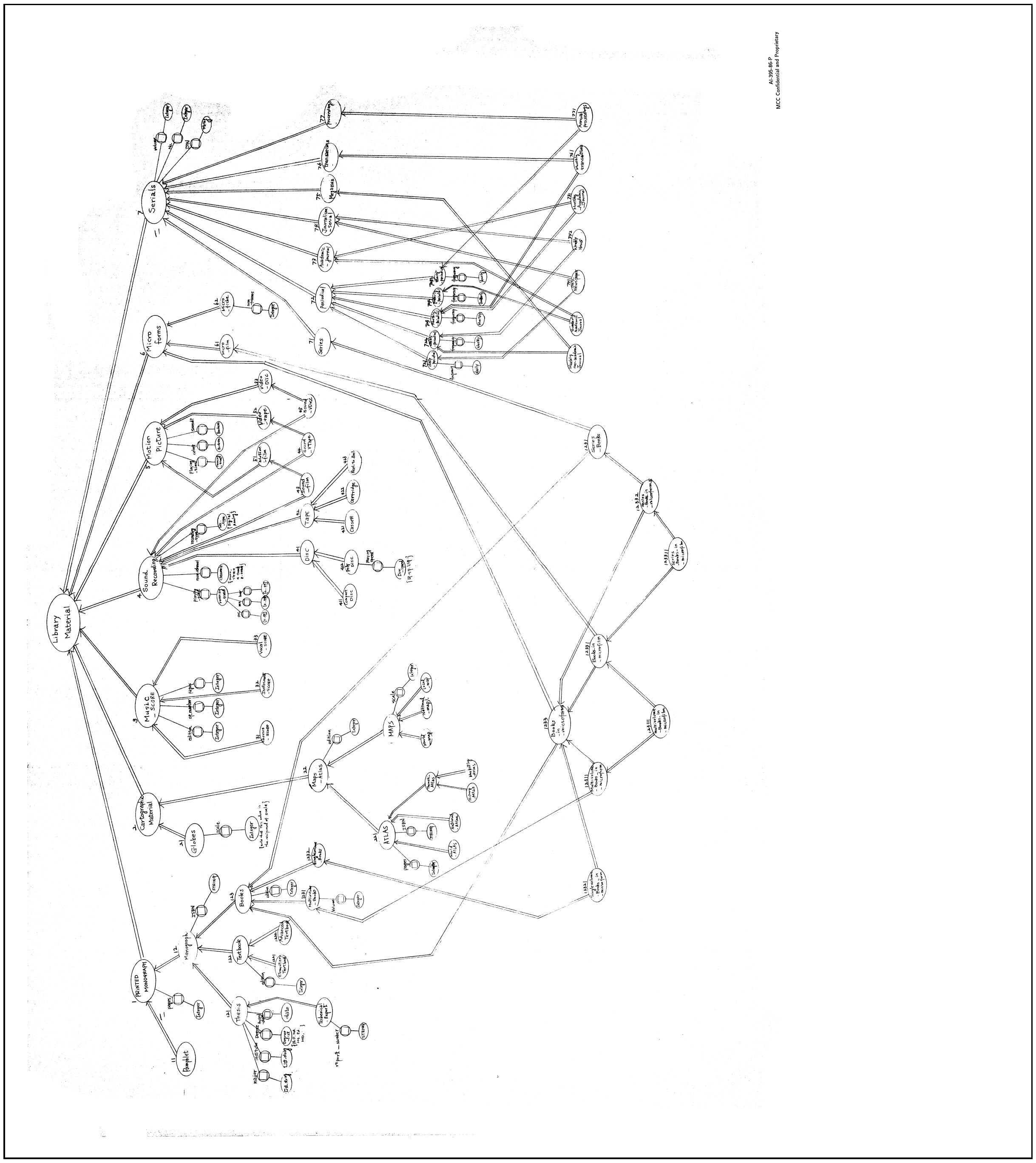

Figure 3: Hierarchy by Subject Class

Once more, *general_reference* cannot be specified with any specific subject, and therefore simply declared to be a subclass of *library_material*:

```
            general_reference < library_material.
```

These classes can also be divided into more specific subclasses. For example, the class of pure science material can be divided into following subclasses having more specific subjects; for example,

```
            mathematics_material
            = pure_science_material(subject => mathematics).
```

Still further, these classes divide into even more specific subclasses, *ad lib*.

Thus, subclasses can be derived as desired to make the structural knowledge base of BABEL a truly static inference system. For example, we can combine two classes, *artificial_intelligence_material* and *programming_language*, as follows:

```
     ai_programming_language_material
     = artificial_intelligence_material(subject => ai_programming_language).

     ai_programming_language_material
     = programming_language_material(subject => ai_programming_language).
```

In this way, we can realize multiple inheritance in a class hierarchy. Indeed, the conceptual hierarchy allows class coercion by subject value unification. For instance

```
     X = linguistics,
     X = artificial_intelligence,
     Y = book(subject => X).
```

becomes

```
     Y = book(subject => natural_language_processing).
```

4.1.3 User Class Hierarchy

The *user hierarchy* is yet another part of the structural knowledge base in BABEL. The class *library_user* is placed at the top of this hierarchy. Subclasses include *faculty_user*, *staff_user*, and *student_user*. The class *student_user* has two subclasses, *graduate_student_user* and *undergraduate_student_user*. All ground values denoting individual user records are placed at the bottom of this hierarchy.

The class *library_user* has several attributes to identify each user. Those are *ss_number* (social security number), *name*, and *address*. One more attribute, *fine*, keeps a record of library fines incurred by library users—elements of this class.

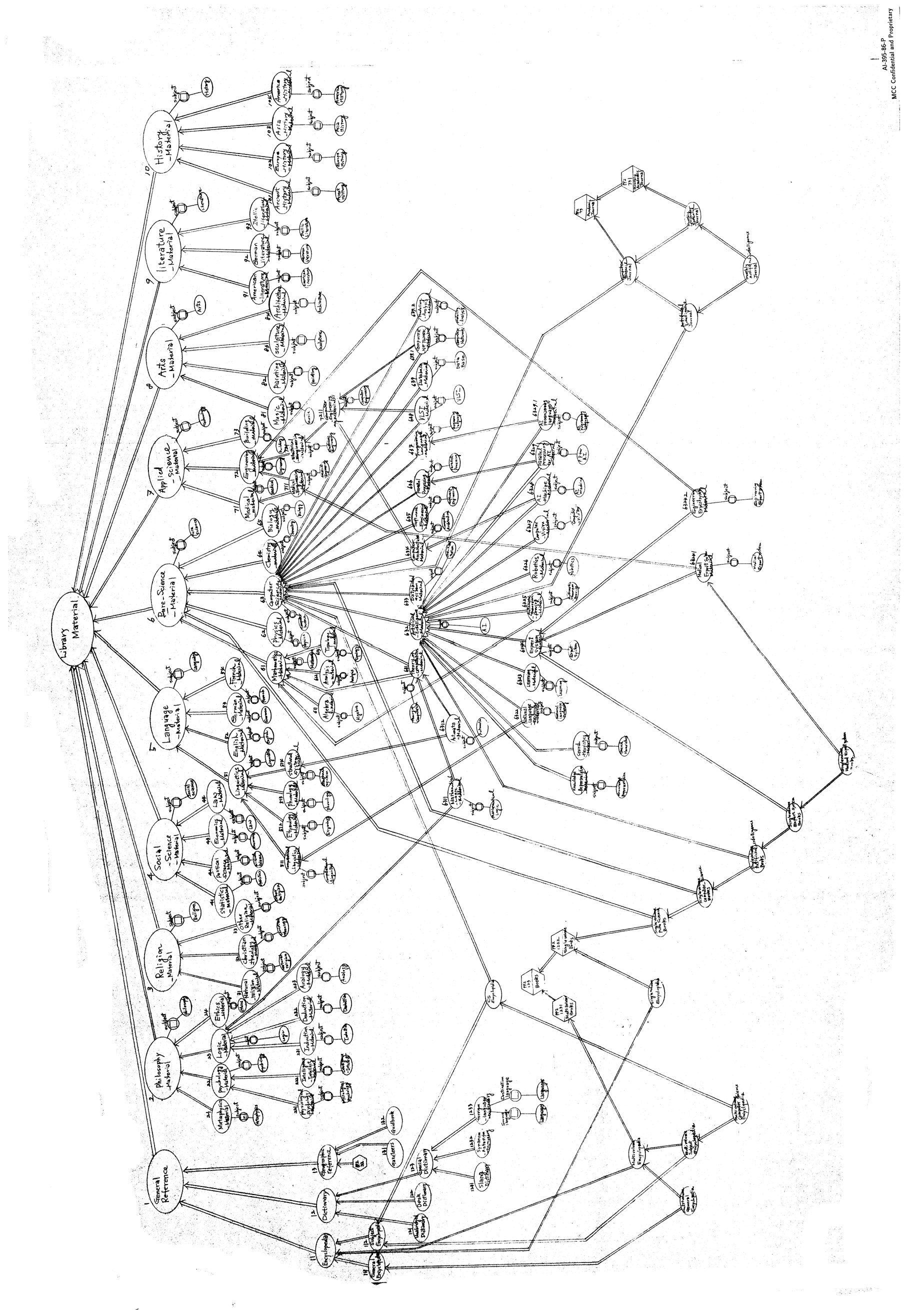

Figure 4: Conceptual Term Hierarchy of Subjects

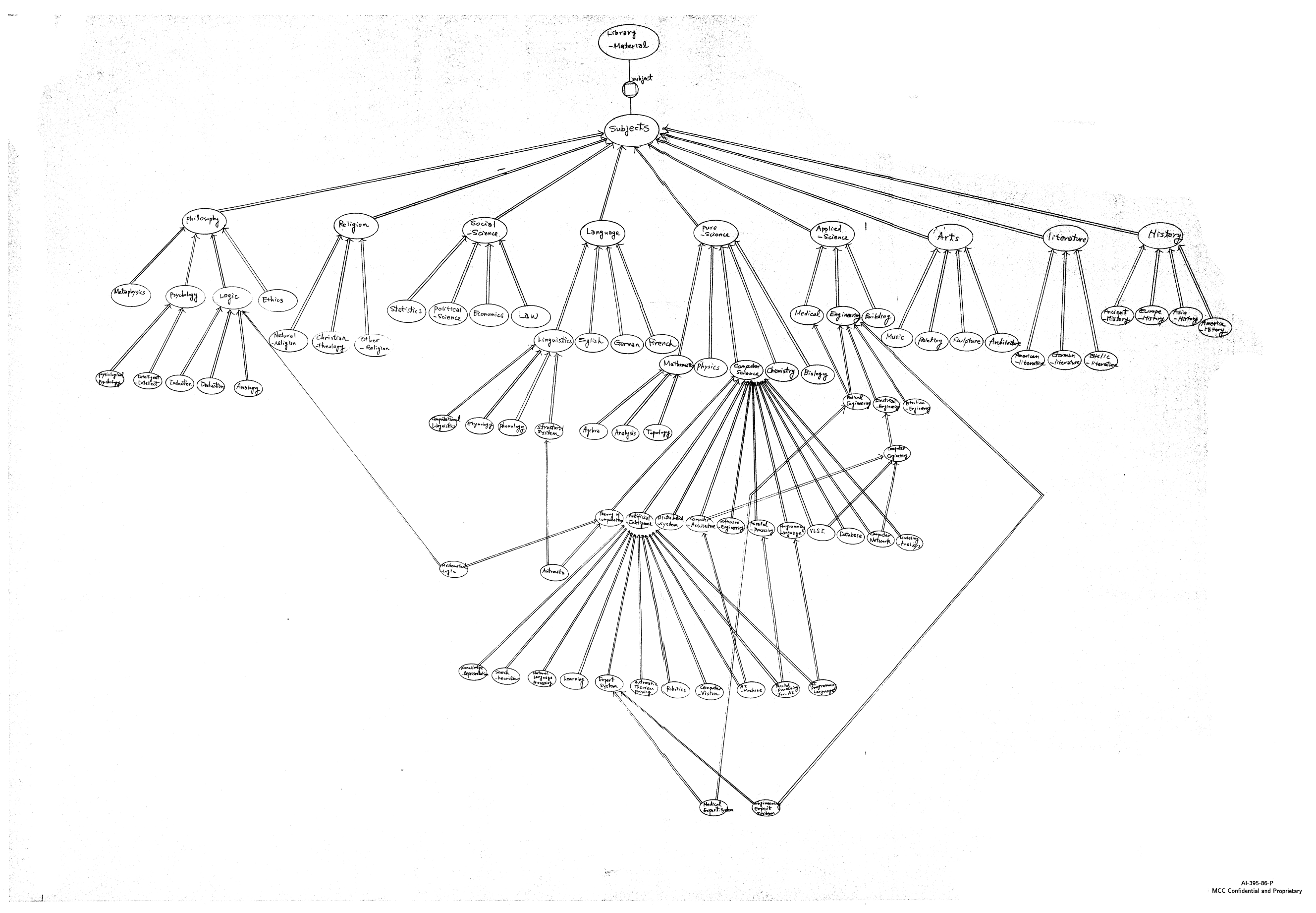

Figure 5: User Hierarchy

4.2 Attributes in Classes of Library Material

4.2.1 Representation of Object States

At this point, we must emphasize a key observation pertaining to the pragmatic use of static *vs.* dynamic information.

Most of the information about an item such as a specific book instance in BABEL, is static. Indeed, author, title, publication, and all such information is not time dependent, and thus is adequately represented in LOGIN as structures suffering no side-effects (attribute/value pairs). However, during the course of its existence in BABEL, such a library item also evolves through a time-dependent maze of states which describe its status at particular points in time. Thus, whether a book is on shelf, borrowed, recalled, *etc.*, is obviously going to determine the behavior of queries about it. Hence, such dynamic information must somehow be side-effected as the BABEL item evolves in time.

Let us take as an example the particular BABEL object: the book *b101* that we have seen before. Let us consider the problem of representing its state showing whether it has been checked out. Having an attribute/value pair such as

```
            b101(check_out => boolean).
```

whose boolean value would be subject to destructive assignments as the book's state changes is obviously not so clean.

The ideal solution would be provided by augmenting Prolog (and thus LOGIN) with a *pointer* type. Assuming such an expression of the form $\uparrow(b101)$ to denote the address of *b101*—now a purely static object—one could thus keep a record of the dynamic information in a dynamic table, *checked_out*, which can be side-effected by assert/retract sequences. For example,

$$check\_out(\uparrow((isbn \Rightarrow \text{`0-86576-004-7'})))$$

will succeed if it has been recorded that *b101* is currently on loan. The reason for desiring to use a pointer to *b101* as opposed to *b101* itself is obviously to avoid copying the whole *b101* structure into the *check_out* table. Clearly, this solution presents the advantage of separating cleanly static and dynamic information about BABEL objects. Unfortunately, this not implementable in Prolog where such pointer expressions are missing. This leads us to our compromise. Of course, this solution is provisional until more versatile addressing primitives are eventually implemented for LOGIN.

At any rate, we built into LOGIN a special type symbol *record__key* which is simply syntactic sugar for a newly generated type symbol (and thus incomparable with any other type symbol other than itself and $\top$). The Lisp programmer may see it as `(gensym)`. Such a symbol is then used as a unique key into a dynamic table which records the changes of state of whatever BABEL item has it as the value of a uniquely corresponding state attribute. Naturally, the value actually generated stays invisible to the user. Thus, provided that *library_material* has the attribute/value pair `check_out => record__key`, the query[7]

```
      ... , library_item(b101(check_out => X)) , recorded(X,yes,_) , ...
```

will succeed only if *b101* is currently recorded as being on loan.

[7]The predicate $record(Key, Value, Reference)$ succeeds with the side effect of entering (asserting) in the Prolog fact (hash) table the pair $\langle Key, Value\rangle$ and sets *Reference* to its memory address. The companion relation $recorded(Key, Value, Reference)$ succeeds if a unifiable corresponding pair was recorded.

4.2.2 Library Object Attributes

In BABEL, each class of the physical hierarchy shares eleven common attributes. Therefore, they can be defined at the highest level of the hierarchy—*i.e.*, for *library_material* (see Figure 6). Those attributes are:

- *title*—title of a library item. It is a list of words.
- *l_responsibility*—list (conjunction) of authors or editors for most works.
- *s_responsibility*—set (disjunction) of authors or editor for most works.
- *subject*—subject category of the item. It has one of the classes in the subject hierarchy as its value.
- *call_number*—has a *lc_number* (library congress number) of the item as a value. *lc_number*, in turn, has several attributes. Those are *c_letter* (category letter), *f_digit* (first digit), *s_digit* (second digit), and *cuttering* of the call number. For example, the call number "QA 76.55 s77" can be represented as

  ```
  (call_number => (c_letter => 'QA',
                    f_digit => 76,
                    s_digit => 55,
                    cuttering => 's77'))
  ```

- *publisher*—an attribute for publisher information. It has a value of type *publish*. The type *publish* has, in turn, several attributes: *publisher_name*, for the name of the publisher, and *address* of the publisher.
- *date_of_pub*—publication date of the item. It has three attributes: *day*, *month*, and *year*.
- *language*—idiom in which the work is published.
- *lend_info*—has *lend_tab* (loan table) as a value type. This loan table *lend_tab* has several attributes:
  - *library_use_only*—record key used to record whether the item may be checked out.
  - *lend_period*—record key used to record the lending period.
  - *check_out*—record key used to record whether the item is currently on loan.
  - *checking_time*—record key used to record when the item was last loaned.
  - *return_time*—record key used to record when the item is due.
  - *ss_number*—record key used to record the social security number of the borrower.

  The attributes, *checking_time*, *return_time*, and *ss_number*, have no meaningful value if the value recorded by *check_out* is *no*.
- *recalled*—record key used to record whether the item has been requested by another user.
- *recall_info*—has a record key used to record a *recall_tab* as a value. The class *recall_tab* has three attributes: *type*, which shows the type of recall request—*recall* or *search*; *ss_number*, which is the social security number of the requesting party; and *more*, which shows other recall requests (if there are multiple requests).

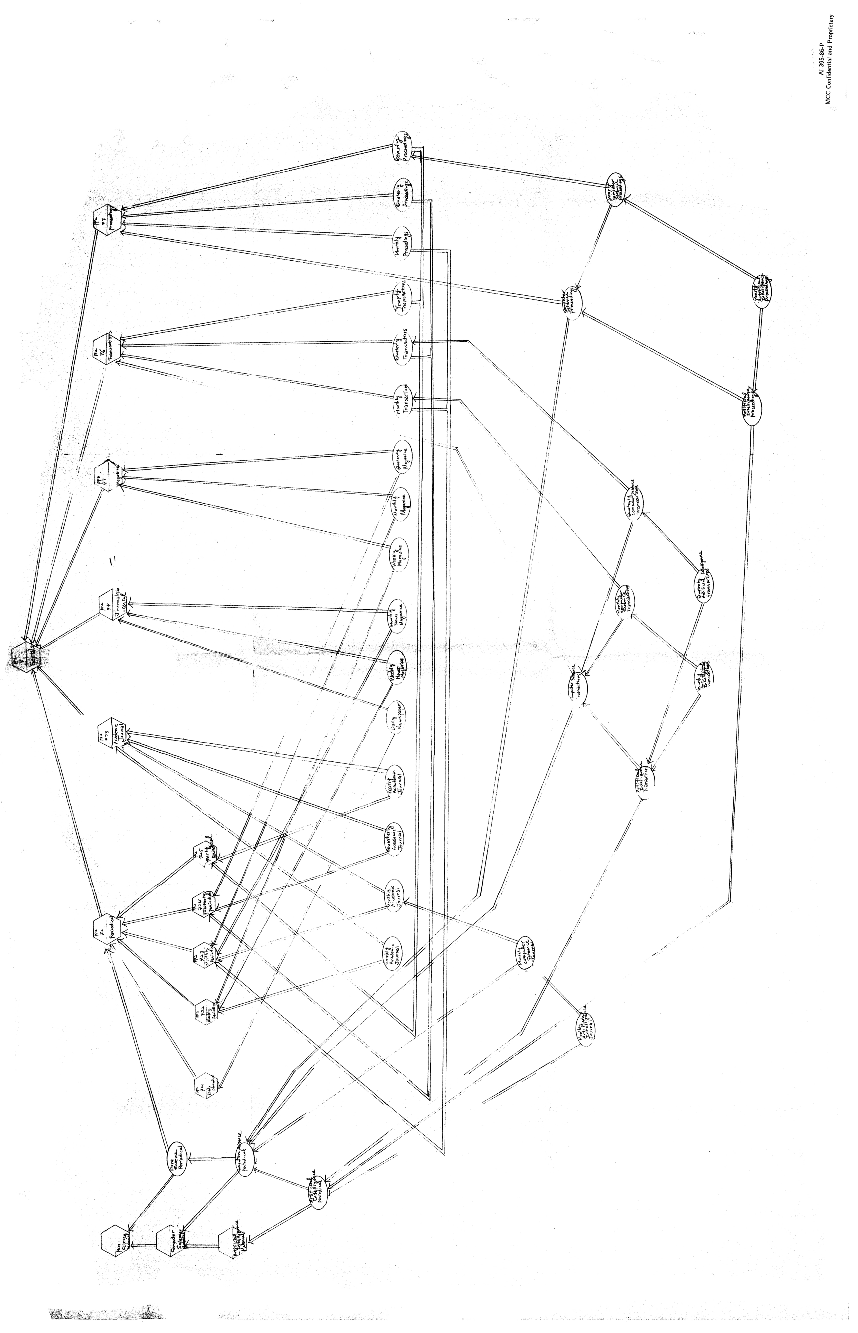

Figure 6: Attributes in Library Material

- *reserve_info*—has a record key used to record the value *reserve_tab* to show whether the item is reserved for special use (*e.g.*, a course). The value appears only if the item is reserved. The class *reserve_tab* has five attributes: *instructor*, *course_name*, *course_number*, *semester*, and *year*.

These eleven attributes are common to all library items. There can be more attributes, depending on the specific class of a library item. For example, every item in the class *monograph* has an attribute *isbn* (International Standard Book Number), while items in *serials* have *issn* (International Standard Serial Number).

BABEL demonstrates an example of its ability to infer by its flexible handling of the attributes pertaining to *statement of responsibility*. Notice that we have two such attributes. One is *s_responsibility*, the other is *l_responsibility*. The information contents of these attributes are the same. However, the internal structures are different. The attribute, *s_responsibility*, is defined as a *disjunctive* class of authors, while the attribute *l_responsibility* is defined as a list of authors. The reason to keep two different attributes for the same information is purely for efficiency. For example, a book item, *b101* say, is asserted to be a *library_item* as follows:

```
b101
= book(
    title => 'The handbook of artificial intelligence',
    s_responsibility =>
         (type => author,
          author => {(fname => F1: 'Avron',
                      lname => L1: 'Barr') ;
                     (fname => F2: 'Edward',
                      mname => M2: 'A.',
                      lname => L2: 'Feigenbaum')},
    l_responsibility =>
         (type => author,
          author => (name => (fname => F1,
                               lname => L1),
                     co_name => (name => (fname => F2,
                                          mname => M2,
                                          lname => L2)))),
    call_number => (c_letter => 'QA',
                    f_digit  =>  76,
                    s_digit  =>  65,
                    cuttering => 'b77'),
    subject => artificial_intelligence).

library_item(b101).
```

When a user wants an item by many authors, but knows the name of only one, BABEL can process the request through the two attributes *s_responsibility* and *l_responsib-ility*. The first is used to verify efficiently (indeed by simple match as opposed to search[8]) the *existence* of the author in the set of the item's authors (disjunction of authors), and the second is used to keep *all* authors (conjunctive list of authors) and to be able to list the names of all authors because we cannot enumerate authors with a disjunctive class. The following user query, although somewhat incomplete, gets a successful response without backtracking:

[8]Actually LOGIN encodes a type hierarchy when it compiles the hierarchy so that it can check types in one matching operation without searching the entire type hierarchy [Aït-Kaci 87].

"Show me a book, written by Feigenbaum, of which subject is computer science"

```
query
:- library_item(book(s_responsibility =>
                         (author => (lname => 'Feigenbaum')),
                     subject => computer_science,
                     title => X)),
   show(X).
```

To implement the above example in Prolog, we need to keep the name of authors as a list of names. For example,

```
/* Fact */
book(/* the title */
     'The handbook of artificial intelligence',
     /* the list of the last names of authors */
     ['Barr', 'Feigenbaum'],
     /* the call number */
     'QA76.65 b77',
     /*  subject */
     artificial_intelligence).

/* Query */
query
:- book(X, Name_list, _, Y),
   is_a(Y, computer_science),
   member('Feigenbaum', Name_list),
   show(X).
```

As demonstrated, in Prolog, the first literal in the query $book(X, Name\_list, \_, \_)$ will succeed with any item in the relation *book*. Then it will check if the subject of the item can be regarded as *computer_science* using *is_a* inference rules which was defined in section 3. Then it will check if *'Feigenbaum'* is in the list of last names of the authors. If any of above two checkings fails, then it backtracks and checks the next item. It might require backtracking through *all* the books in the library.

5 Transaction and Query Managers

BABEL would not be much of an experiment without some sample application programs. In this section, we discuss the transaction and query managers of BABEL. The query manager serves the user of the library, and the transaction manager serves the staff.

5.1 Transaction Manager

In its present state of conception, BABEL manages five basic transactions for the librarian:

- Maintaining records of current library material and users;[9]

[9]In its current status, implementation of BABEL does not support on-line data insert and delete operations which require dynamic updates of class hierarchy.

- Check out and return operations;
- Recall request processing;
- Reserve request processing;
- User fine calculation and notice generation.

These are, of course, not meant to be as realistic as possible—*e.g.*, few libraries (hopefully) fine without repeated warnings!

5.1.1 Checkout and Return operations

When a user wishes to borrow a book, the librarian requests information from a terminal at the lending desk by call number. If the book is cleared for lending (not reserved or previously requested), its status is updated during the lending transaction as follows:

```
 check_out :-
    /* Read call number of an item */
    get_call_number(LC_number : lc_number),
    /* Get the record of the item using the call number*/
    library_item((call_number => LC_number,
                  lend_info => (lend_period => Lend_Period_Key,
                                checked_out => Checked_out_Key,
                                checking_time => Checking_time_Key,
                                ss_number => SS_number_Key,
                                return_time => Return_time_Key),
                  recalled    => Recalled_Key,
                  recall_info => Recall_info_Key)),
    /* Read the social security number of the user */
    get_ss_number(SS_num),
    /* If somebody recalled the item, and */
    ( recorded(Recalled_Key, yes, _)
      -> /* if the recall person is the borrower */
         ( recorded(SS_number_Key, recall_tab(ss_number => SS_num), _)
           -> remove_recall(LC_number)
           ;  (write('This item is recalled by another user.'),
               fail))
      ;  true),
    /* Set Check_out to yes */
    putvar(Checked_out_Key, yes),
    /* Set user ss_number to SS_number */
    putvar(SS_number_Key, SS_num),
    ( more_to_check_out
      -> check_out
      ;  true ).
```

The return transaction is as follows. Any fines are assessed at this time:

```
return :-
  /* Read call number of an item */
  get_call_number(LC_number : lc_number),
  /* Get the record using the call number */
  library_item(Item : (call_number => LC_number,
                       lend_info => Linfo:(checked_out => Checked_out_Key,
                                           ss_number => SS_number_Key,
                                           return_time => Return_time_Key),
                       recalled => Recalled_Key)),
  get_current_time(CU_time),
  /* Now, Checked_out becomes 'no' */
  putvar(Checked_out, no),
  /* get expected return time from the record with record key */
  recorded(Return_time_Key, Return_time, _),
  /* If it is over the due date, */
  ( late_return(CU_time, Return_time),
    -> /* calculate and make a fine notice */
       process_fine(Item, SS_num),
    ;  true),
  ( more_to_return
    -> return
    ;  true).
```

5.1.2 Recall Request Operation

If a book is not on shelf, a user can request it through one of two operations. In the first, a *search* request, the book is located and held for one week. In the second, a *recall* request, the requested book is already on loan. A notice is sent to the current borrower that the book has been requested. When it is returned, the librarian sends the person who made the request a notice that the book is waiting. The algorithm follows.

```
add_recall :-
    /* Read call number */
    get_call_number(LC_number : lc_number),
    /* Get the record using the call number */
    library_item(Item(call_number => LC_number,
                      recalled => Recalled_Key,
                      recall_info => Recall_info_Key)),
    /* Read recall informations, */
    get_recall_info(Recall_tab: recall_tab),
    /* put the recall informations at the end of the recall list */
    put_end(Recall_info_Key, Recall_tab),
    /* Now, Recalled becomes 'yes' */
    putvar(Recalled, yes)),
    ( more_to_recall
      -> add_recall
      ;  true).
```

5.1.3 Reserve Request Operation

When an instructor needs certain material to remain available in the library for students during the period of a given course, the librarian enters that information using the following algorithm.

```
reserve :-
  /* Read call number of the item */
```

```
get_call_number(LC_number : lc_number),
/* Get the record of the item using the call number */
library_item((call_number => LC_number,
              lend_info => (lend_period => Lend_Key,
                            library_use_only => Luse_Key),
              reserve_info => Reserve_info_Key)),
( library_use_only
  -> /* Classify if the item is requested  for library use only */
     putvar(Luse_Key, yes)
  ;  /* or not for library use only */
     putvar(Luse_Key, no))
/* Read desired loan period, and */
get_period(Period),
/* put the value to the attribute lend_period */
putvar(Lend_Key, Period),
/* Read reserve informations, and */
get_reserve_info(Reserve_Tab),
/* put it to the attribute reserve_info*/
putvar(Reserve_info_Key, Reserve_Tab),
( more_to_reserve
  -> add_reserve
  ;  true).
```

5.2 Query Manager

The query manager in BABEL is an interactive query generator using menu operation. Once it is activated, it shows possible options which may be used for searching through BABEL which constitute the user's interactive choice. From this choice, the query manager generates a formal LOGIN query, and executes it. There are various kinds of search queries, which can be classified according to the key value to be used. Typical examples are,

1. Show by call number
2. Show by title
3. Show by statement of responsibility, *i.e.*, author, editor, *etc.*
4. Show by temporal, *i.e.*, publication date, *etc.*
5. Show by ISBN for books
6. Show by ISSN for serials
7. Show by subject
8. Show by physical description

The user may choose one or more keys to find an item. For example, if the user chooses 1, for call number, then the query manager reads the call number, *e.g.*, 'QA 76.66 b77', and generates the query such as,

```
library_item(X : (call_number => (c_letter  => 'QA',
                                  f_digit   => 76,
                                  s_digit   => 66,
                                  cuttering => 'b77')),
show_item(X).
```

If the user chooses 3 and 7, then the system gets the name of author, *e.g.*, *'Feigenbaum'*, and the category of subject, *e.g.*, *computer_science*. Then, the generated query looks like

```
library_item(X : computer_science_material
                    (author => (lname => 'Feigenbaum'))),
show_item(X).
```

The generalized unification operation in LOGIN provides two invaluable built-in search strategies: *focusing* computation only on relevant domains of objects, and *intersecting* such domains. Such are indeed very effective for navigating through the knowledge base of BABEL. Both features are performed in exactly same manner—through ψ-unification— but one can use them differently.

For example, let us suppose that a user requests a book on both computer science and linguistics written by Robert F. Simmons. There are two possible ways to request it. The first:

```
library_item(X : computer_science_book
                    (author => (fname => 'Robert',
                                mname => 'F.',
                                lname => 'Simmons'))),
library_item(X : linguistics_book),
show_item(X).
```

and the second

```
X = computer_science,
X = linguistics,
library_item(Y : book(author => (fname => 'Robert',
                                 mname => 'F.'
                                 lname => 'Simmons'),
                      subject => X)),
show_item(Y).
```

Through unification steps in LOGIN, the first query becomes:

```
library_item(X : nlp_book(author => (fname => 'Robert',
                                     mname => 'F.',
                                     lname => 'Simmons'))),
show_item(X).
```

The domain of X which is *computer_science_book* and *linguistics_book* is intersected down to *natural_language_processing_book*. This is *focusing* the domain of objects.

On the other hand, the second query becomes:

```
library_item(Y : book(author => (fname => 'Robert',
                                 mname => 'F.',
                                 lname => 'Simmons'),
                      subject => natural_language_processing)),
show_item(Y).
```

In this case, the variable X, which first becomes *computer_science* through unification, is *intersected* with the term *linguistics*, and finally becomes *natural_language_processing*. In this way, intersection of types keeps a finer and finer focus on relevant solutions.

In BABEL, we can use either of the above queries, thanks to the hierarchy of physical library material classified by subjects together with the conceptual subject hierarchy.

Operationally, however, there is a big difference between the two queries. In the first query, we can reduce the search space by *focusing* the domain of objects, but not in the second query. In the latter case, LOGIN will visit each record in the class *book* to check whether the value of *subject* is *natural_language_processing*.

6 Conclusion

We have presented a prototype library expert system, BABEL, written in LOGIN. By using ψ-terms, rather than first-order terms, LOGIN provides effective methods to represent FRAME style knowledge structure while keeping full expressive power of Prolog.

The strategy of ψ-term unification algorithm with order-sorted Horn logic [Smolka 87] provides efficient way to express set theoretical operations. Every set theoretical operations can be expressed in unification operation in LOGIN. Thus, it is possible to use a *set-at-a-time* operation in LOGIN. For example, we can use *assert* to assert the fact whether a library material is reserved. If an AI faculty requests to reserve all *artificial_intelligence_material*, then we can use *set-at-a-time* operation by asserting the whole set of *artificial_intelligence_material* is reserved.

```
LOGIN ?- assert(reserved(artificial_intelligence_material)).
yes
LOGIN ?- reserved((title => 'The handbook of artificial intelligence')).
yes
```

As in above example, since any item which is an artificial intelligence material succeeds to unify with *artificial_intelligence_material*, we can assert a set at a time without asserting every items in AI as reserved.

In its comparison with its potential implementation in Prolog, BABEL in LOGIN seems to provide the following advantages.

1. The knowledge base taxonomy is easy to use and maintain because of the following features:
   - Since attributes of a class inherit to its subclass, it allows default value.
   - Every attribute is represented as a pair consisting of a label and a value, as opposed to a position number and a value. Thus, the order of arguments in a ψ-term is transparent to the programmer. As a result, the number of attributes in two ψ-terms does not need to be the same to unify the ψ-terms.
   - LOGIN directly supports new programming features like semantic domain definitions and domain *focusing* and *intersecting*.
   - The above features enable a user to get without much overhead a response from queries containing incomplete information.
2. LOGIN code is also likely to be better than Prolog's for several reasons:

- Many deductions can be done at the unification level rather than at the resolution level. For example, the unification strategy using disjunctive classes, as described in Section 4.2, significantly reduces the number of environment changes and backtrackings.
- By adding more restrictions to each argument in a query, one can reduce the domain of the search space.

In summary, intrinsic features of LOGIN look very promising for dealing efficiently with the rich taxonomic information found in information systems such as BABEL.

In the course of developing this experiment, ideas for real-life software development in a sophisticated language like LOGIN have emerged. Such an example is the integration of a user-specifiable inheritance operation on specific attributes. A mild example in the form of a self-description facility is being added to LOGIN for BABEL where definition strings are related by prefix ordering as opposed to subclass ordering. Extensions thereof may be string or regular expression matching, *etc.*

In conclusion, we shall certainly admit that more work needs to be done to determine fully the effectiveness of LOGIN in a real application. To make its advantages concrete, we need to be careful in implementing an engine for LOGIN and its extensions so that the overhead in performing a generalized unification algorithm does not outweigh the efficiency gained.

References

[Aït-Kaci 86a] Aït-Kaci, H., "An Algebraic Semantics Approach to the Resolution of Type Equations." *Theoretical Computer Science* **45**, 1986, pp. 293–351.[online[10]]

[Aït-Kaci 87] Aït-Kaci, H., Boyer, R., Lincoln, P., and R. Nasr, *The Efficient Implementation of Object Inheritance.* MCC Technical Report AI-102-87. Microelectronics and Computer Technology Corporation, Austin, TX. July, 1987.[online[11]]

[Aït-Kaci 86b] Aït-Kaci, H. and R. Nasr, "LOGIN: A Logic Programming with built-in Inheritance." *Journal of Logic Programming* **3**(3), October, 1986, pp. 185–215.[online[12]]

[Aït-Kaci 88] Aït-Kaci, H., Nasr, R., and J. Seo "BABEL: A Base for an Experimental Library." *Proceedings of ACM SIGIR 11th Conference on Research and Development in Information Retrieval*, Grenoble, France, June 13–15, 1988, Presses Universitaires de Grenoble, pp. 175–190.[online[13]]

[Bobrow 77] Bobrow, D. G., and T. Winograd, "An overview of KRL, a Knowledge Representation Language." *Cognitive Science* **1**, 1977, pp. 3–46

[Goldstein 77] Goldstein, I. P. and R. B. Robert, "NUDGE, a Knowledge-Based Scheduling Program." *Proceedings of 5th IJCAI*, 1977 , pp. 257–263.

[Goldberg 80] Goldberg, A. and D. Robson, *Smalltalk80: The Language and its implementation*, Addison-Wesley, 1980.

[10] http://hassan-ait-kaci.net/pdf/tcs-86.pdf
[11] https://hassan-ait-kaci.net/pdf/encoding-toplas-89.pdf
[12] https://hassan-ait-kaci.net/pdf/login-jlp-86.pdf
[13] https://hassan-ait-kaci.net/pdf/babel.pdf

[Gorman 78] Gorman, M. and P. W. Winker, (Eds.), *Anglo-American Cataloguing Rules*, 2nd ed., America Library Association and Canada Library Association, 1978.

[Minsky 75] Minsky, M., "A framework for representing knowledge." In P. Winston (Ed.) *The psychology of computer vision*, McGraw-Hill, 1975, pp. 211–277

[Smolka 87] Smolka G., and H. Aït-Kaci, *Inheritance Hierarchies: Semantics and Unification*. MCC Technical Report AI-057-87. Microelectronics and Computer Technology Corporation, Austin, May 1987. (To appear in 1989 in C. Kirchner (Ed.) *Journal of Symbolic Computation*, Special Issue on Unification.)[online[14]]

[Stefik 83] Stefik, M., D. G. Bobrow, S. Mittal, and L. Conway, "Knowledge Programming in LOOPS: Report on an Experimental Course," *Artificial Intelligence*, Fall 1983, pp. 3–14.

[14] https://hassan-ait-kaci.net/pdf/jsc-89.pdf