A SYSTEM FOR INTERFACING LIFE WITH DATABASE AND PERSISTENT STORAGE

Sanjay Gupta
B.Tech., Indian Institute of Technology, Delhi, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
November 1996

0-612-24145-9

APPROVAL

Name:	Sanjay Gupta
Degree:	Master of Science
Title of thesis:	A system for interfacing LIFE with database and persistent storage
Examining Committee:	Dr. Rob Cameron Professor Computing Science, Chairman
	Dr. Hassan Aït-Kaci, Senior Supervisor
	Dr. Jiawei Han, Supervisor
	Dr. Frederick P. Popowich, Examiner
Date Approved:	November 25, 1996.

Abstract

LIFE is a functional logic programming language extended with object-oriented concepts(sub-typing and inheritance). The objects in LIFE are extensible, complex, and partially ordered. LIFE can be viewed as a combination of functional, logical and imperative programming paradigms. The combination of these three different programming paradigms in LIFE provides powerful high-level expressions and facilitates specification of complex constraints on data-objects. Therefore it is ideally suited for applications in natural language processing, document-preparation, expert systems, and so on. These applications rely on large amounts of data and will require database technology for efficient storage and retrieval of data. Keeping this in mind, we extend LIFE with database interfaces for object-oriented and relational data.

These interfaces are used to store LIFE facts and persistent terms. The reverse problem, conversion of relational data into LIFE as ψ-terms, has also been addressed in this thesis. We give an algorithm to generate LIFE facts from relational data. Efficacy of these approaches has been studied using real word problems arising in Geographic Information Systems and Information Retrieval Systems.

dedicated to my parents Sh. Ramautar Gupta and Shmt. Kaushalya Gupta

Acknowledgements

It is with my deepest appreciation that I acknowledge my advisor, Professor Hassan Aït-Kaci for his financial support, advice and continued encouragement. He has always made himself available to answer my questions and discussion of my ideas. I am grateful to him for his guidance, patience and encouragement in my research in making it a most rewarding experience. It has been a privilege to have been his student.

I would also like to thank the members of my committee who all made useful comments about my work. Special thanks to Hassan Aït-Kaci, Jiawei Han and Fred Popowich for patiently reading my thesis and providing helpful suggestions.

My heartfelt thanks to Dayaram Gaur for reading through several of my thesis drafts providing meaningful suggestions to improve the legibility of the thesis. He also deserves credit for some of the research work in this thesis. I am also grateful to Pattabhiraman for proof reading the first draft of the thesis. Martin Vorbeck provided help in the initial technical discusion of this research.

I also would like to acknowledge many of my house-mates, office-mates and friends who have made Vancouver a special place to live: Sumeet Bawa, Pinaki Mitra, Dayaram Gaur, Vikas Gupta, Pattabhiraman Thyagarajan, Latha Thyagarajan, Martin Vorbeck, Harry Grewal, Roman Bachik, Subho Chatterjee, Ajay Srivastav, Ashish Pimplapure, Mark Mezofenyi, Graham Finlayson, Aamir Hussain and Taranjit Singh. My former room-mates Sumeet Bawa and Pinaki Mitra made my initial days in Vancouver much happier - Sumeet Bawa for his true friendship, watching movies with him, numerous trips to pub for coffee with him, his drinking sessions and jokes, and Pinaki Mitra for his humor. Martin Vorbeck and Ashish Pimplapure provided good company as my office mates and diversion from work. Pattabhiraman's one liners about life was a good source of entertainment and Latha's cooking of idlees and dosais reminded me of my hometown Madras. My current room-mates Dayaram Gaur and Vikas Gupta provided good company and made the writing process enjoyable - for their cooking, for playing snooker when the stress of thesis writing was too much and for Vikas for bearing with me to watch star trek on TV. Special thanks to Dayaram Gaur for his encouragement in making the process of writing the thesis easier.

Special appreciation to my brother Rajesh and sisters Madhu and Vinu for their confidence in my abilities, for their encouragement to get on with my life and my work in my difficult times. Finally I wish to express the deepest gratitude to my parents for the love and affection bestowed on me and their moral support. This thesis is dedicated to my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Considerable research effort has been directed recently towards combining databases and programming languages [14, 21, 23, 27]. While each of the systems on their own provides considerable utility in their respective application domains, a large number of applications exists that need the functionality provided by both the systems. From a programming language perspective the need was felt because

- Application programmers would benefit enormously from being able to manipulate persistent data [21] (data that outlived the execution of the program) just as they manipulate non-persistent data, removing ad hoc facilities for data translation and long term storage.

- Applications typically work on large sets of data that do not fit in main memory and cannot be efficiently handled by the programming language. Furthermore existing databases need to be accessed by a programming language.

- An application handling large amount of data uses only a small part of it during a program run. Files are generally read and written as a unit; therefore access and updates for large files is slower. Performance can be improved if only data is retrieved and updated incrementally as needed.

In the database community, the need was felt because

- Programming languages like C++, LIFE, offer alternative data-modeling [1, 11, 14, 35, 33] capabilities for systems like NLP, CAD, document retrieval systems, software engineering, hypertext data, etc, which deal with complex object structures.
- It is recognized that object-oriented programming languages enable appropriate modeling of problem domains and reduce the effort involved in translating the application model to the model of the implementation language.
- Logic programming as a database query language [14] offers greater expressivity for queries and constraints than other languages. It is widely recognized that we need to combine the query processing part of logic programming systems with the efficient access techniques of DBMS's.

Aït-Kaci et.al [6] have conceived a new programming language, LIFE (logic, inheritance, functions and equations), that combines three different programming paradigms: logic programming, functional programming and object-oriented programming, providing a powerful formalism for many different applications [10, 11] that include natural language processing, expert systems, intelligent document retrieval systems, etc. LIFE provides a basic data structure called the *psi*-term, which neatly supports frame-style knowledge structures [13]. The knowledge-base can be structured as an inheritance hierarchy in LIFE [3, 10] and together with the ψ-term unification algorithm, provides group-related processing tasks such as answering set queries, discriminating between objects, finding similarities between objects, etc. In addition, the intermingling of relational and functional expressions in LIFE [8, 9] in a declarative manner allows powerful high-level expressions and complex constraints on data-objects.

Moreover the ψ-term data-model can represent arbitrarily complex objects, with no constraints on the size or the structure. This is especially useful for applications which manipulate large sets of data, where the types of data-elements in a set need not be the same (for example, document retrieval systems, hypertext data, etc).

LIFE cannot deal with the size, amount or the distributed nature of the data in such advanced information systems, and will need extensions to manage the secondary storage requirements and communication needs of the application.

The objective of this thesis is:

- to provide database management facilities for LIFE, combining them in as "seamless" a manner as possible. The system should be efficient enough to avoid having an adverse effect on performance.
- to demonstrate LIFE's capabilities as a knowledge and database manipulation language. Two applications are described and their performance analyzed.
- provide the ability to use data stored in a relational database from a LIFE program.

After a brief introduction of LIFE's basic data-structure and unification algorithm, we will illustrate with some of the advantages of LIFE's knowledge and database modeling capabilities and the data-objects in LIFE that have to be stored in an external database.

1.1 LIFE overview

In this section, the main concepts of LIFE will be summarized. The basic data-structure in LIFE is called the ψ-term, which is a useful extension to the first order term as in Prolog [12]. A first-order term in Prolog consists of either a constant, a variable or a term of form $s(a_1, \ldots, a_n)$, where s is the *functor*, and a_i's (which are arguments of the term) are first-order terms. A first-order term with no arguments is called a *constant*. Variables are denoted by strings beginning with _ or an uppercase letter, while a functor symbol starts with a lowercase letter.

One of the main drawbacks in the above representation of Prolog term is that arguments in the term need to be identified by the position at which they occur in the term. In a ψ-term, symbols called *labels* or *features* are used to identify the arguments. This extension helps make writing programs easier, avoid programming errors and improve the readability of the programs. The order of the arguments in a ψ-term is immaterial and the arity of the term (number of arguments in the term) is not fixed.

1.1.1 ψ-term Data Structure

Informally, a ψ-term is characterized by specifying a type constructor, called the *root sort* and a set of attributes($\langle label, value\rangle$ pairs), where the order of these pairs is immaterial. An attribute is defined by specifying its name and domain. The attribute-values can be other ψ-terms, both primitive (such as integer, real, boolean, sorts or string) and non-primitive ones.

Example 1.1 *An example of a ψ-term is:*

```
researcher ( last_name       ⇒ string,
             date_of_birth   ⇒ date ( day   ⇒ integer,
                                      month ⇒ integer,
                                      year  ⇒ integer
                                    ),
             specialization  ⇒ research_area,
             works_at        ⇒ organization_name
           ).
```

This is an example of a **researcher** type, consisting of a root sort **researcher** and four sub-ψ-terms denoted by attribute labels *last_name, date_of_birth, specialization and works_at.* This ψ-term describes an object **researcher** consisting of four record fields.

Variables and Tags in ψ-terms:

Unlike a Prolog term, a LIFE term can be conceived as an arbitrary graph structure which may include cycles. This is made possible by using variables as *tags* (references) to structures within a ψ-term. In a Prolog term, a variable can appear only at the leaf node of a term. However in a LIFE term, variables can appear anywhere in the ψ-term structure. This allows variables to be used as tag symbol for structure sharing (called *coreference constraint*) within a ψ-term.

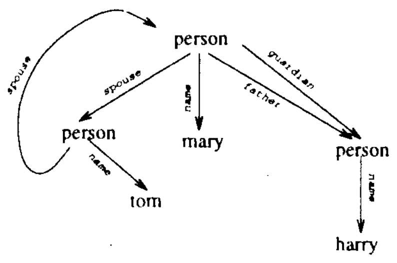

Figure 1.1: Graph representation of the ψ-term in example 1.2

Example 1.2 *An example of a ψ-term using tags for coreference constraint is:*

```
P:person ( name ⇒ ''mary'',
           father ⇒ X:person(name ⇒ ''harry''),
           guardian ⇒ X,
           spouse ⇒ person(name ⇒ ''Tom'', spouse ⇒ P
         ).
```

The tag symbol X refers to the fact the father and the guardian of the above person is the same, while the tag P defines a cyclic coreference constraint that the spouse of a spouse of a person is the same person.

LIFE provides a powerful type mechanism and extends first-order terms with a subtype relation on the symbols occurring in the terms. The following subsection explains the notion of subtyping in LIFE.

1.1.2 Types and Inheritance

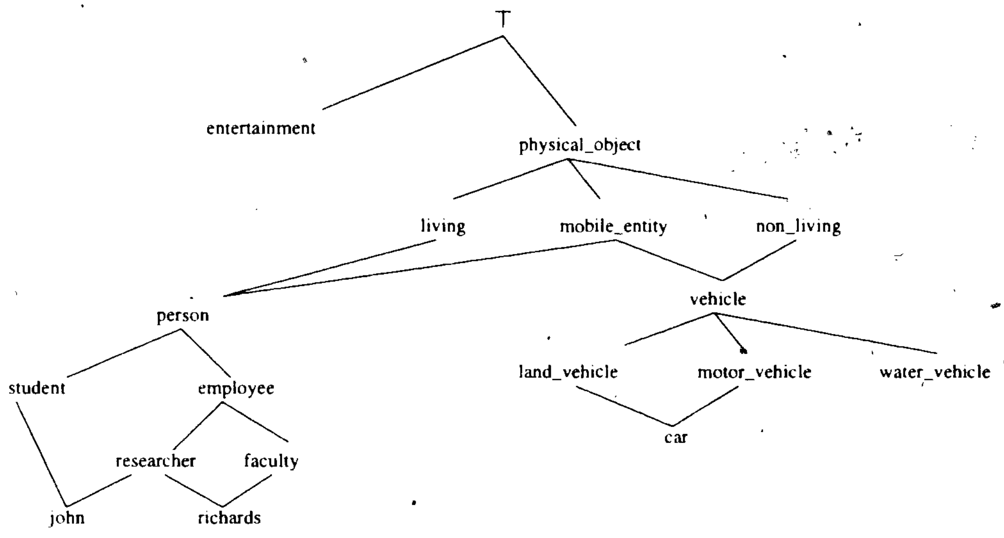

Figure 1.2: Type hierarchy in LIFE

The type symbols in a LIFE program are partially ordered with the relation $\leq$. Fig 1.2 shows an example of the partial order on the type symbols in a LIFE program. For instance it defines that the **researcher** is a subtype of **employee** (defined in LIFE program as: `researcher` $\leq$ `employee.`) which in turn is a subtype of **person**. The type hierarchy includes two special types $\top$ (top symbol) encompassing all the types and $\perp$ (bottom symbol, not shown in the figure) encompassed by all other types in LIFE.

We can also define multiple type relations in a single statement such as `person := {student;researcher}`. This is equivalent to stating **student** $\leq$ **person** and **researcher** $\leq$ **person** in LIFE.

In LIFE, no distinction is made between types and values. Thus all integer values such as 1 are treated as subtype of a built-in type **int**. Another example of built-in type is **string**, and all strings s are subtype of **string**. Figure 1.3 shows the builtin types and the subtype relationship between them in LIFE.

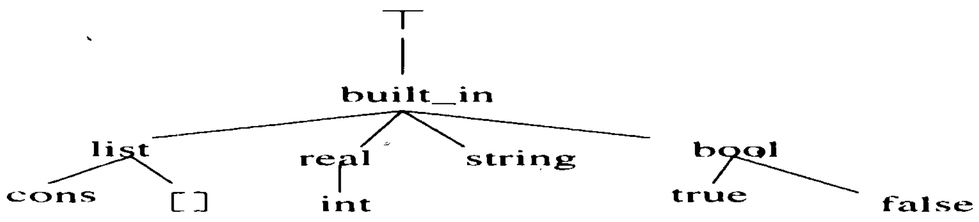

Figure 1.3: Built-ins type hierarchy in LIFE

We can also define new types and attach properties to it such as attributes and constraints. For example:
`::book(title ⇒ string).`
states that a **book** has a **title** attribute which should be a string. Any ψ-term instance in a LIFE program whose root sort is a **book** is unified (unification is explained in section 1.1.3) with the above definition. The attribute values could be functional expressions as well (functions in LIFE are explained in section 1.1.4), in which case these functions are evaluated before unification.

We could also attach constraints to the sorts. An example of constraints attached to a sort is:
`::s(name=>X:string) | print(X).`
The symbol " — " is pronounced as "such that" and the constraint that follows is of the form of a definite clause body.

The subtype relation and attribute specification can be done in a single definition such as:

`employee ≤ person(salary ⇒ int).`

which states that an **employee** is a **person** with an additional attribute *salary*.

Each subtype inherits the attributes of its parents. For example if the definitions of **person** and **employee** are:

```
::person ( last_name     ⇒ string,
           date_of_birth ⇒ date ( day   ⇒ integer,
                                  month ⇒ integer,
                                  year  ⇒ integer)
         ), and

employee ≤ person(works_at ⇒ organization_name).
```

The **researcher** type of example 1.1 can be redefined as: researcher ≤ employee(specialization ⇒ research_area) inheriting the attributes identified by the labels *last_name*, *date_of_birth* and *works_at* from **person** and **employee** class definitions.

1.1.3 Unification of ψ-terms

To make use of the inheritance mechanism in LIFE, the unification algorithm of first-order terms is generalized. Two ψ-terms p and q are unifiable iff:

- the corresponding root sorts have a common sub-type in the type hierarchy.
- their sub-terms (identified by same label name in both the terms) unify recursively.
- Missing attributes do not prevent unification. If a label is present only in one of the terms i.e an attribute present in one term is missing in the second term, the attribute value for the second term defaults to the

special symbol $\top$ (the least defined type). $\top$ subsumes all other type symbols in a LIFE program and unifies with all type symbols.

Example 1.3 *Consider the type hierarchy in Fig 1.2. In this example we have the greatest common subtype of* employee *and* student *is* john, *the greatest common subtype of* researcher *and* faculty *is* richards, *and the greatest common subtype of* land_vehicle *and* motor_vehicle *is* car.

Now given the two terms:

```
student( student_id ⇒ int,                    employee( salary ⇒ int,
         fname ⇒ A:string,                              fname ⇒ joe
         lname ⇒ A:string,                              lname ⇒ joe
         supervisor ⇒ researcher,       and             supervisor ⇒ faculty,
         owns ⇒ motor_vehicle)                          owns ⇒ land_vehicle)
```

unifying the two terms results in

```
john( student_id ⇒ int,
      salary ⇒ int,
      fname ⇒ A:joe,
      lname ⇒ A:joe,
      supervisor ⇒ richards,
      owns ⇒ car).
```

1.1.4 LIFE Program

A LIFE program is a collection of ψ-terms denoting either:

1. a *definition* terminated by a period, and can be either a:

   (a) *predicate definition*: `H :-` $B_1, \cdots, B_n$.

   (b) *function definition* `H` $\rightarrow$ `B.`

   (c) *type definition*: `reseacher` $\leq$ `employee.`

Here H, B and B_i are ψ-terms.

2. a *query* terminated by a question mark such as: $B_1, \cdots, B_n$?. A LIFE *query* is a list of *goals* B_i's to prove. A *goal* is a LIFE ψ-term whose root symbol is a predicate name. A top-down/left-right SLD resolution is used to prove a goal.

The names of functions, predicates and types in LIFE begin with a lowercase letter, while variables begin with _ or an uppercase letter. Such variables are local to the clause in which they occur.

Predicate Definition

A LIFE predicate is of the form `H :-` $B_1, \cdots, B_n$. LIFE predicates are defined in the same way as in Prolog comprising of one or more clauses. Clauses comprise of a head and a body. Head H consists of a single ψ-term, and the body B_i comprises of multiple ψ-terms, and either the junctor "," (logical *and*) or ";"(logical *or*). A single clause predicate containing only head and no body is a *fact*. A *rule* comprises of head and a body, where the head succeeds if the body succeeds.

Function Definition

Another major extension to Prolog is the addition of functional capabilities in LIFE. A function definition in LIFE is of the form H $\rightarrow$ B. It comprises of a head ψ-term H and a body ψ-term B. B is the value returned if the function evaluation succeeds.

Functions can be built-in functions such as $(+, -, *)$ or user defined functions illustrated in the next example.

Example 1.4 *We define a function* fact *that computes factorial of a number.*

```
fact(0) → 1.
fact(N:int) → N*fact(N-1).
```

A function definition is a collection of functional rules. The rules of a function are evaluated in the order in which they are asserted. Unlike predicates, if any one of the rules of a function fires (the first functional rule whose head ψ-term unifies with the query) there is no backtracking to the next rule. The first functional rule to fire is one whose head ψ-term unifies with the query.

A functional rule whose head ψ-term unifies with the query then is evaluated if the arguments of the functions are sufficiently instantiated, i.e if the arguments of the function call are subtype of arguments in function definition. It *residuates* (delays the evaluation of function) if the arguments are not sufficiently instantiated. The residuating function keeps track of its arguments, and evaluates when all arguments are sufficiently instantiated. The concept of functional residuation provides a form of concurrent programming in LIFE.

Type Definition

We saw the type definition concept in section 1.1.2. We can also attach coreference constraints or functional constraints to types. For example to state that a square is a rectangle whose length and width are equal, we can write it as:
`square ≤ rectangle(length ⇒ X:int, width ⇒ X).`
and an example of type definition with functional constraint is:
`::rectangle(length ⇒ X:int, width ⇒ Y:int, area ⇒ X*Y).`

Disjunctive Terms

It is also possible to concisely represent ψ-terms in LIFE, by means of disjunctive terms. A disjunctive term is an expression of the form $\{A_1; \ldots; A_n\}$, where A_i's are ψ-terms. It creates a choice point returning the value A_1 on the first instance. On backtracking it returns the next value A_2 and so on. For example the predicate assertion likes(sam, {apples;oranges;peaches}) is equivalent to asserting the predicates likes(sam, apples), likes(sam, oranges) and likes(sam, peaches). In both the cases the query **likes(sam,X)** will result in X=apples, and on successive backtracking give X=oranges and X=peaches.

Global variables

LIFE also provides the concept of *global variable* A global variable in LIFE is a variable which is accessible throughout the program, unlike a normal variable whose scope is limited to the clauses in which they occur. Global variable names begin in lowercase letter and share the same name space as that of predicates, functions and types. An example of global variable declaration is `global(count)?`. The symbol *global* is a built-in predicate which declares *count* to be a global variable. Global variables behave like normal variables, except that they are visible throughout the program.

1.2 LIFE: A Knowledge and Database Manipulation Language

Large scale knowledge bases require more intelligent processing than current Data Base Management Systems (DBMS) offer. LIFE as a pure logic programming language constitutes an attractive query language. However, LIFE is more than just a logic programming language, since it offers features and embodies a data model turning it into an intelligent information processing system. This stems from the fact that the data model can be represented as a ψ-term. In this section we give an informal presentation of how LIFE can be used as a knowledge-base and database manipulation language.

The knowledge component can be described in LIFE by different knowledge representation techniques:

- *Facts and Rules* represent declarative knowledge, which may be factual, for example **faye** is the child of **sue** (child(faye, sue)), or inferred from rules, such as X is a parent of Y, if Y is a child of X (parent(X,Y) :- child(Y,X)).

- *Frame style*: Frames describe a collection of objects with common properties, consisting of a list of attributes($\langle label, value \rangle$ pairs). For example:

$vikranth(type \Rightarrow sportscar, roof \Rightarrow convertible, doors \Rightarrow 2, wheels \Rightarrow 4)$ encodes the knowledge that vikranth is a sports car with a convertible roof and two doors.

- *Inheritance* refers to the concept that an object takes on the properties characterizing its parent object. It is a powerful mechanism for encoding and storing knowledge economically. The common information and behavior can be captured at the topmost node in a classification hierarchy, rather than at every single relevant node.

The inheritance mechanism in LIFE together with its ψ-term unification mechanism provide efficient expression of set-theoretic operations [10]. To illustrate this, we consider a simple example of an *e-mail address* database relation of persons, the organization they work in and their area of expertise:

Table 1.1: An address database

Name	*Status*	*Organization*	*Specialization*	*e_mail*
viraj bais	post_doc	sfu	cg,ai	vb@sfu.ca
sand meyer	student_researcher	ubc	ai	sm@ubc.ca
joe peter	student	ubc	database	jp@ubc.ca
steve rich	student	sfu	mathematics	sr@sfu.ca

This example (Table 1.1) focuses on one particular advantage of LIFE, that of inheritance structure in LIFE, a powerful mechanism to encode knowledge very economically.

Example 1.5 *The* e-mail address *relation in Table 1.1 can be represented as* facts *in LIFE as follows:*

```
address ( name ⇒ (fname ⇒ ''viraj'', lname ⇒ ''bais''),
          status ⇒ post_doc, organization ⇒ sfu,
          specialization ⇒ cg;ai, e_mail ⇒ ''vbais@sfu.ca''
        ).
```

A knowledge-base for the database, encoded as a type-hierarchy, is shown in Fig 1.4 using the set-inclusion relationship between types. For example the types **researcher, faculty, consultant** are classified as sub-types of **employee**. Multiple inheritance is also supported: for example **professor** is a sub-concept of both **researcher** and **faculty**.

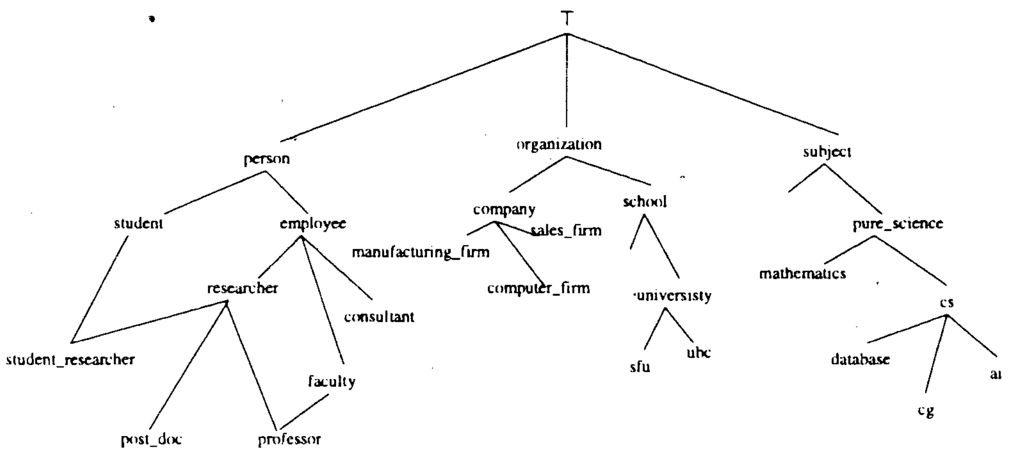

Figure 1.4: Concept hierarchy for address database

Sample query

Given the set of *address facts* and the knowledge-base coded as inheritance hierarchy, consider the following query:

> "Retrieve from the database, the names and the e_mail-addresses of the *researchers* working in the *computing science* field."

The query could be posed in LIFE as

```
address ( name ⇒ (fname ⇒ X, lname ⇒ Y),
          status ⇒ researcher, specialization ⇒ cs,
          e_mail ⇒ Z
```

)?

Output

For the sample query above, we get the tuples in table 1.2

Table 1.2: Sample query output

Name	*e_mail*
viraj bais	vbais@sfu.ca
sand meyer	sm@ubc.ca

The knowledge-base of Fig 1.4 enables intelligent retrieval of information. Although the address database did not literally contain the word *cs*, owing to the fact that *cg*, *ai* are classified as sub-types of *cs*, the query retrieved e-mail addresses of persons working in *cg*, *ai* as well.

1.3 Thesis Organization

Chapter 2 provides an overview of efforts aimed at applying database technology to programming languages, and identifies the techniques best suited for the LIFE system. Chapter 3 contains a description of LIFE-RDBMS interface we have implemented. The interface takes a coupling approach to store LIFE facts in SQL databases. LIFE also supports data objects (persistent ψ-terms) which can contain references to other data objects. In chapter 4 we explore the issues related to storing these inter-object references. We also describe the architecture of the long term storage of persistent terms in LIFE that was implemented. We also address the issue of accessing data stored in a relational database. In chapter 5 a clustering algorithm is presented that extracts a hierachical categorization of relational facts in LIFE. Chapter 6 presents two applications in LIFE and their DBMS store performance and results. We conclude this dissertation with a discussion in chapter 7.

In the following sections we will introduce the next three chapters, relational database interface to LIFE, storage of persistent data and the reverse problem: that of translating relational data into ψ-terms.

1.3.1 LIFE to RDBMS Interface

A typical LIFE program will consists of a large set of facts, such as the address database in the example 1.5. The size of data renders the use of main memory as a storage device impractical. The data has to be stored on the disk. For this reason, we have implemented a system coupling LIFE with relational databases, representing arbitrarily complex objects in LIFE as flat relations to be stored in the external database. In contrast to systems offering predicates for storage and retrieval of facts from an external database, in our system the access to the data in the DBMS and the existence of a database under LIFE is transparent to the user.

The implementation is based on the theory proposed by Holsheimer in [1]. Based on the theory [1] a clustering mechanism based on the type hierarchy of LIFE for clustering the *facts* (localization of data) has been implemented to improve access time. The theory [1] also provides a sound mechanism for selective data retrieval of the facts stored in the database. We make use of this mechanism for retrieval of facts from RDBMS in LIFE. This provides a filtering effect, retrieving all potentially unifiable facts with the query and providing a small resolution-set for full unification. The information about the past interaction with the relational database is cached in a compact and efficient way. This reduces the number of calls to the database, as it never repeats any query, subsumed by the queries in the past. Techniques for optimization of data retrieval by means of dealing with overlapping queries, and with intelligent detection of intersections have been implemented.

The above mentioned theory can only handle types of single inheritance. In the implementation modifications were made to this theory to handle multiply inherited types in LIFE. The theory [1] also cannot handle facts containing coreference constraints. Such coreference constraints are common in applications like NLP. We also show how to handle such coreference constraints. Another shortcoming in the theory [1] was handling of variables in goal. We present a solution for it and further optimize the data retrieval by exploiting the type information stored in the relational schema. Chapter 3 reports on our implementation of a LIFE-RDBMS interface. As an example application to test this interface, a bibliographic database was implemented and its performance results are presented in chapter 6.

1.3.2 Persistent Terms in LIFE

Manipulating the identity of objects is natural in a language with pointer types such as C++. *Persistent ψ-terms* in LIFE provide a similar effect by extending the "value-based" semantics of Prolog with the ability to access the identity of data objects.

A persistent ψ-term (example 1.6 and example 1.7) unlike a 'normal' ψ-term retains its value on backtracking, undergoing an unrestricted update similar to assignment in an imperative language. It is changed by a nonbacktrackable infix assignment operator $\twoheadleftarrow$ and cannot be modified through unification. The terms can be viewed as a global database (a set of graphs) with handles on certain nodes.

A persistent term can be assigned to a local/global variable using the assignment $\twoheadleftarrow$. Any further modification of the term persists on backtracking, except when one backtracks before the point at which the variable was assigned a persistent term. In that case the variable retains its old value, and the persistent term is no longer accessible through this variable.

Persistent terms can also be assigned to a *persistent variable*, in which case the term is always accessible (unlike persistent term assignment to local and global variables). A persistent variable declaration is like a global variable, and its scope too extends throughout the program. The built-in predicate *persistent* is used to declare a persistent variable. Only a persistent term can be assigned to a persistent variable.

Example 1.6 *This example illustrates persistent variables.*

```
persistent(library_item)?
library_item <<-book(title => ''computer graphics'',
                      author => ''richard'')?
```

If a new assignment is made to the variable library_item, the new value is retained on backtracking.

A persistent ψ-term can contain references to other persistent ψ-terms as well. In life the subterms of a ψ-term are accessed using the "." operator. For example library_item.title gives access to the subterm "computer graphics" in the term assigned to library_item in example 1.6. The next example illustrates the structure sharing between two persistent terms.

Example 1.7 *An example of sharing references between persistent terms:*

```
persistent(book101, book102)?
book101 <<- book ( title => ''computer graphics''
                   author => person(name => richard,
                                    e-mail => rich@sfu.ca))?
book102 <<- book(title => ''computer algorithms'')?
book102.author <<- book101.author?
```

Figure 1.5 shows the two persistent terms assigned to the persistent variables book101 and book102 in a graph form and the structure sharing occurring between them.

This graph data model represent's complex object structures in a natural way, modeling objects as nodes (with attributes representing their properties) and relations between them as edges. But the notion of persistency in LIFE is limited to a program life-time. Long-term persistency is needed for LIFE. In chapter 4, we explore the issues relating to persistency of data in programming languages, describe our implementation of the persistent ψ-terms in LIFE,

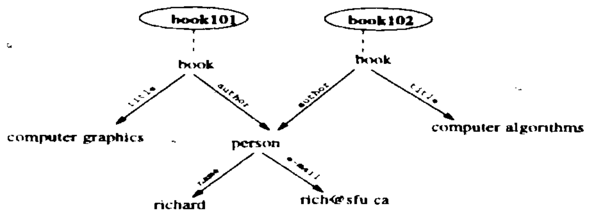

Figure 1.5: Structure sharing between two persistent ψ-terms

and present performance results. An application in Geographic Information Systems (GIS) was implemented using persistent terms in LIFE. Chapter 6 contains a description of the GIS application along with performance results of long term persistency in LIFE.

1.3.3 Reverse Compiler

We would also like to provide the ability to use data available in existing relational database from LIFE programs. A simple solution for this would be to provide a predicate that takes in an SQL string and returns the results as ψ-terms. The relational database provide simple data structures, which take up large storage space in primary and secondary memories. LIFE provides an elegant type mechanism, which can be used to represent the relational facts economically and improve the performance.

Example 1.8 *As an example we consider a binary relation* rel *containing*
rel(a,r), rel(a,s), rel(a,t), rel(b,r).
rel(b,s), rel(b,t), rel(d,q), rel(d,r).
rel(d,s), rel(d,t), rel(e,q), rel(e,r).
rel(e,s), rel(e,t).

The relations are decomposed into two rectangles (Rect1 and Rect2) as shown in figure 1.6. The two rectangles of this decomposed relation could then

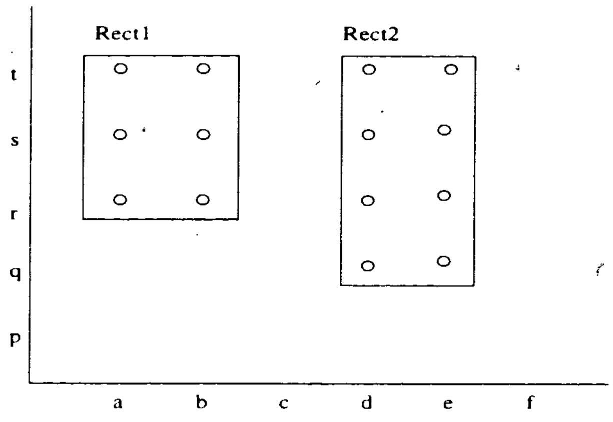

Figure 1.6: Rectangular decomposition of relation *rel* of example 1.8

be concisely represented in LIFE as facts:
`rel(ca1, ra1).` % Rect1 in figure 1.6
`rel(ca2, ra2).` % Rect2 in figure 1.6

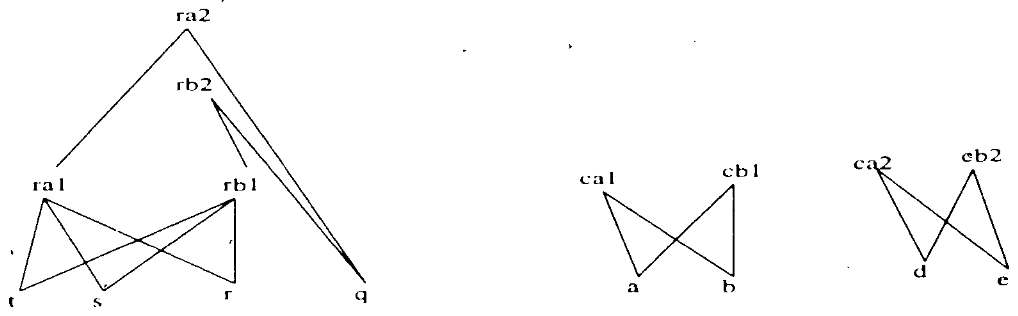

Figure 1.7: Hierachical categorization of relation *rel* of example 1.8

A type hierarchy for the sorts **ra1**, **ca1**, **ra2** and **ca2** and for the symbols in the relation *rel* is shown in figure 1.7. The constraints attached to these sorts are:

```
::  X: ra1 | X=rb1.
::  X: ra2 | X=rb2.
::  X: ca1 | X=cb1.
::  X: ca2 | X=cb2.
```

The constraints force the enumeration of relation *rel* facts in LIFE. The search space for the relation *rel* is reduced in LIFE (2 as opposed to 14 in the original database) and would result in significant performance gain.

The algorithm to decompose binary relations as rectangles is given in chapter 5. This will provide us with a hierarchical categorization of relations in LIFE.

Chapter 2

Combining DBMS and PL

2.1 Introduction

In recent years considerable research has been directed at combining programming languages and databases. The combined use of object-oriented languages, logic programming languages, etc and database management systems offers a powerful problem solving architecture for a wide range of applications. Database systems provide an environment offering facilities for creation and maintenance of large, long-lived collections of data; these facilities include efficient access, data security and transaction processing. However, the expressive power of the languages provided within database systems is limited. Programming languages on the other hand provide facilities for procedural control, data and functional abstraction, but lack built-in support for any of the database features. Thus, a coupling of programming languages and DBMS is natural.

2.1.1 Database Systems

Programming languages such as object-oriented languages and logic languages have been used to enhance the capabilities of databases. Such languages are an attractive proposition for several reasons:

Alternative data model: Complex data modeling capabilities are required by systems like NLP, CAD, document retrieval systems, software engineering, hypertext data etc, which cannot be done easily in relational databases. Programming languages such as C++ provide appropriate modeling of problem domains and ease the effort involved in translating the resulting application model to an implementation.

Query language: Logic programming as a database query language [14] offers a greater power for expressing queries (such as recursive queries) and constraints than SQL used in relational databases. These languages provide the ability to make inferences over large volumes of data. DBMS users such as managers and specialists, have a high degree of domain knowledge but little patience to familiarize with programming language concepts. For such users high level languages offer a better model to interact with the database.

Optimization: Declarative languages like LIFE and Prolog provide a powerful formalisms to implement a flexible query optimization component. The declarative nature of such languages enables rapid prototyping and testing of an idea too.

2.1.2 Programming Languages

Languages like C++ and Prolog often need the support of database features. There are several reasons why such languages need database features:

Persistent data: Sometime programmers need the capability to manipulate persistent data (data that outlived the execution of the program) just as they manipulate non-persistent data. This will reduce the effort involved by the programmer for data translation. Such systems can also support, sharing large amounts of data among multiple users in distributed computing environments.

Efficiency: An application handling large collections of data, uses only a small part of it during a program run. Since files are generally read and written as a unit, access and updates are slower for larger files. DBMS on the other hand offer powerful and efficient features for access and modification of large volumes of data. Performance can be improved, if data can be off-loaded to the database system for more efficient, and possibly parallel access.

Handling large data sets: Applications typically work on large sets of data that do not fit in main memory, and cannot be handled by the language.

Accessing existing databases: Sometimes existing databases need to be accessed by the programming language.

Heterogeneous databases: There is always the need to couple existing heterogeneous systems into a cohesive environment without sacrificing the privacy and/or independence of the participating systems. High level languages like LIFE have the capability to provide a common conceptual view of the distributed data in heterogeneous databases.

2.2 Alternatives in Design

There have been several approaches to integrate database and programming languages, all of which provide a synergistic combination of the two technologies. The two major approaches are:

1. *Coupling* stand alone database management systems and programming languages (mainly logic programming systems). The overall architecture combines a general-purpose programming language as the front end with a DBMS back end. It preserves the independence of the end systems.

2. *Integration* constitutes designing a single system in which the database functionality is incorporated. In such systems no clear distinction exists between front and back end components.

2.2.1 Coupling

This approach [2, 14] tends to maintain the identity of each component. The front-end is a general-purpose programming language and a DBMS is the back end containing facts for front-end reasoning. The programming system is essentially devoted to data manipulation tasks, while the DBMS acts as a server supplying the data to the front end for further processing. Both the database management system and the programming language environment run as separate processes communicating through some channel.

There have been many proposed and implemented systems for coupling a logic programming language (such as Prolog) with a relational DBMS [15, 19]. These systems have been broadly divided into two categories, based on the degree of coupling (loose, tight) [2]:

Loosely-coupled systems

In a loosely-coupled system, the required data is fetched from the external database into the active memory in a prior phase (at program load time) before the work begins on a set of related problems. The DBMS interacts with the programming system PS in a "batch" fashion: maps the required data from the DBMS to the PS data structures at the beginning of a session, and copies back the modified data at the end of a session. The identification of the data required is done for example in Prolog, by providing a meta-interpreter for determining the queries to be evaluated and evaluating them before running the Prolog interpreter.

Tightly-coupled systems

In a tightly coupled architecture, the data is fetched on an as-needed basis. The programming system (PS) may interact with the DBMS at any moment during the problem solving process. The actual computation process activates the interface, dynamically generating database queries and retrieving data for the PS to proceed further. The main issues in such a system is how much data to fetch (prefetching) and whether to cache the data.

A tightly coupled system is generally preferred over loosely coupled systems as it can make use of the actual computation process, to anticipate the data needs of the application better. This is especially true for large systems, which requires less pollution of main memory for better performance. On the other hand for small systems, it may be better to load the data in one-shot, being as selective as possible during data retrieval. This will avoid the overhead of frequent database access of tightly coupled systems.

2.2.2 Integration

An integrated system consists of designing a single system in which some or all of the database functionality is incorporated. In such systems no clear distinction exists between front and back end components.

There are two possibilities: an existing programming language system is incorporated with database functionality, or new systems are designed from scratch.

Persistent programming languages

A *persistent programming language* (PPL) is a programming language that provides for data to exist beyond the life-time of a program. PPL incorporates the database concept of persistency into its programming model, and provides the ability to manipulate persistent data (data used in successive executions of the program) just as they manipulate transient data. Different programs could also access the persistent data.

Database programming languages

A *database programming language* (DBPL) is similar to a PPL but incorporates additional database features beyond persistency such as bulk data (sets or relations) and object-content based retrievals (queries).

New databases

New databases are designed such as deductive databases and object-oriented databases. The systems are built from scratch, without depending on previously built software, with no previous bad decisions affecting the new design. They typically need developing new data models and algorithms.

2.3 Design choices for LIFE

For the approaches presented in the previous section, the main problems are, how can a system be at the same time:

- reasonably cheap (requiring little or no modification to either LIFE or the DBMS source code if any),
- user friendly (transparency of the data distribution),
- and efficient.

The integration of DBMS into LIFE can take either the coupled approach or the integrated approach. Both would have major differences in complexity, performance and the ease of design and use.

Developmental effort of coupled systems is small and the resulting product has the potential of providing adequate functionality and performance. Efficient integration of the systems is the only issue that needs to be focused upon in such systems.

On the other hand, in an integrated system one can directly use low level functionalities of the DBMS, like relation management in secondary memory, and data access via indices. It is possible to let both tools access the internal structures of the other in ways that coupling cannot allow, offering more opportunities for optimizing and fine-tuning the system. Furthermore, translation to different data formats is avoided, providing a superior performance over coupled systems. The price that must be paid is a need for extensive modification to one or both of the tools to get them to work with each other.

An attractive feature of coupled systems is that they can utilize the existing system with little or no modification a factor critical to the portability of the system. Another factor is the need to use an existing database. As there is always the need to couple existing heterogeneous systems into a cohesive environment without sacrificing the privacy and/or independence of the participating systems.

In the next subsections we evaluate the questions posed so far in the context of LIFE.

2.3.1 Coupling Approach

We choose to use relational database systems for long term storage of LIFE facts. Although LIFE facts are complex objects consisting of cycles and structure sharing (see section 1.1.1), the structure sharing is local to the facts (i.e there is no data sharing between facts). A grouping of facts can be defined, as the facts in LIFE generally constitute identically structured objects. These then can be represented as flat tuples in a relational table (explained in section 3.3).

An object-oriented database could be a better alternative, as it provides similar complex objects and inheritance structures found in LIFE reducing the semantic mismatch between the two. But the theory [1] developed behind LIFE facts storage and retrieval does not make use of any of the features provided by object-oriented databases.

A coupling approach is used because standard relational databases exist and are widely used. They provide for free transaction management and concurrency control facilities to ensure data integrity: allowing sharing of information in a multiuser environment and having a recovery mechanism for stable storage in the event of a crash. In addition to the use of an existing databases, the interface can easily be ported to other RDBMS systems as well.

2.3.2 LIFE as a Persistent Programming Language

For the persistent terms in LIFE, relational database technology is inadequate. As shown previously in section 1.3.2, a major characteristic of such data is that the terms may contain references to other terms. Such references can be dynamically created too, and an attribute *value* may change to a *reference* to another term as shown in example 2.1.

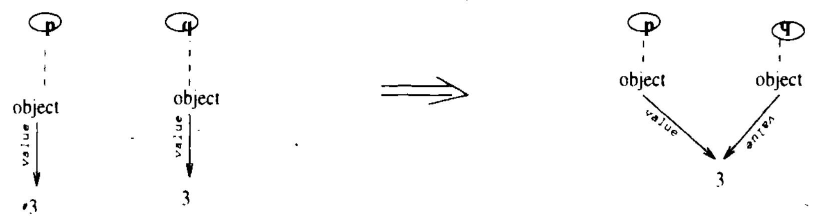

Figure 2.1: Dynamic updates of persistent terms

Example 2.1 *An example of two persistent variables* – p,q. *In this example the value of* b *changes from 3 to a reference to* p*'s value.*
persistent(p,q)?
p $\prec\!\!\prec\!\!-$ *object(value* $\Rightarrow$ *3)?*
q $\prec\!\!\prec\!\!-$ *object(value* $\Rightarrow$ *3)?*
p.value $\prec\!\!\prec\!\!-$ *q.value?*

Figure 2.1 shows the change done in example 2.1 graphically.

These inter-object references are difficult for relational database to handle. An OODBMS is natural for such terms, providing both object-id based retrieval of single object and set-oriented operations. We decided to build an object-store on top of file system provided by the operating system for the sake of portability.

Chapter 3

Coupling LIFE to a Relational Database

3.1 Introduction

As shown previously in section 1.1.4 a typical LIFE program consists of

unit clause (Fact): H.

rule: H :- $B_1, \ldots, B_n$.

type definition: researcher $\leq$ employee.

goal: $B_1, \ldots, B_n$?.

Here H and B's are ψ-terms.

The *logical facts* and *rules* in the form of Horn-clauses, can be separated into Extensional Database (EDB) and Intensional database (IDB) facts. The EDB is simply a large collection of *facts*, while the IDB is built from the EDB by applying rules to it. A fundamental assumption in LIFE has been that the *facts* reside in main memory; for small problems this assumption is not a restriction. However for applications handling large sets of *facts*, LIFE's ability is limited. Typical database applications handle large data sets containing a million facts or more. With such large data sets secondary storage has to be relied upon for processing data, providing an environment where rules (and possibly small sets of facts) are stored in LIFE and (large sets of) facts are stored in a database.

The objective of the LIFE-RDBMS interface is to provide, as efficiently as possible:

- the support for decomposition of complex facts into flat relations.
- transparent retrieval of data-items into LIFE from relational DBMS, using several optimizing techniques.

A LIFE system, which makes use of a relational database for storing and retrieving facts, has been built. The interaction between LIFE system and the database objects is independent of any user support. The interface supports programs written in pure LIFE, and the existence of a database under LIFE is transparent to the user. This is made possible by a *program analyzer* which compiles the original LIFE program into a modified LIFE program, incorporating the original rules and a *data dictionary*, minus the facts which are asserted into the external database. The data dictionary provides transparent retrieval of the facts stored in the external database. The compiled code can also be linked to other applications written in LIFE, requiring no special support.

A relation consists of identically formed objects, and in order to store the *facts* as flat relations, we group the facts into identically formed facts. A grouping of facts can be defined, as the facts in LIFE generally constitute identically structured objects. In order to do so, a subtype order $\leq$ is defined on the facts as well, based on the subtype ordering on type symbols that make up these facts. The facts with similar subtype relationships are grouped together and stored in a relation. These groups, called *qualified segments*, contain identically formed facts. These concepts will be explained and illustrated later in section 3.3.2.

For data retrieval, we use a tight coupling, where facts are loaded on demand. The interface accepts arbitrarily complex goals and return all facts unifying with the goal. The system is based on a two-stage filtering process shown in figure. 3.1. A *qualifier* (data-definition) is defined for each qualified segment (contains facts with similar subtype relationships), which is a generalization of all the facts in the segment. The *qualifier* provides a filtering effect retrieving all potentially unifiable facts for queries and providing a smaller resolution-set for full unification, thereby improving the efficiency of the unification languages. The loading mechanism keeps track of information about the past interactions with the database. Previous queries to the database are cached in a compact and efficient way. This provides a second stage filtering effect minimizing the number of accesses to the database, as it avoids repeating queries, subsumed by past queries to the database.

LIFE memory gets polluted when the retrieved databases facts are cached in main memory. Automatic eviction of database facts is provided when LIFE memory becomes full. An LRU (least recently used fact name) policy is used when evicting database facts from LIFE memory.

3.2 Architecture

The interface between LIFE system and the database can be provided at various levels of transparency [14]:

No transparency In this approach the programmer uses the data manipulation language of the database engine to manipulates objects in the external database. PROSQL [41] takes this approach, providing a special built-in predicate, which takes an SQL query in the form of a Prolog string as its argument. This has the advantage that no translation is needed, and the full extent of SQL can be utilized. A major weakness is that the responsibility of the interaction between the two systems is left to the user. In addition, access optimization options are limited, and also the programmer has to cope with two different languages.

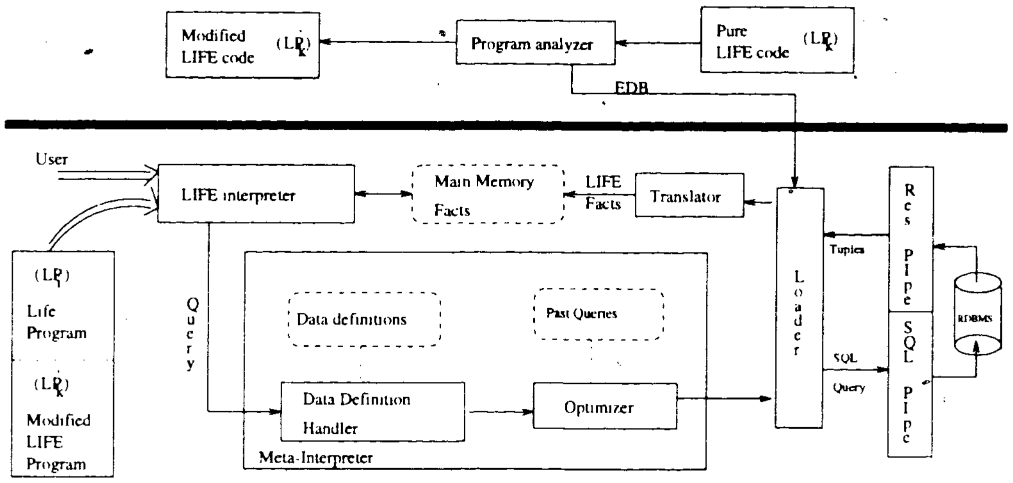

Figure 3.1: Interface architecture

Intermediate transparency Data access, whether they are in main memory or disk-resident, is transparent to the user; there is no need for translating from one representation to another. But the user has to explicitly declare database predicates, which provide the relationship between the two representations.

Full transparency The interaction between the programming system and the database is done, without requiring any user support. The existence of a database is transparent to the user.

We shall use the third approach as it insulates the user from the interaction between LIFE system and the external database. This approach provides support for pure LIFE programs, and the impression given to the user is that of interacting with the LIFE system. This is made possible with the help of a *program analyzer* translating the original program into a modified program, which can recognize database predicates and acts accordingly.

For the rest of the chapter, *database predicates* (facts in LIFE) are LIFE predicates that are stored in the external database. In the forthcoming examples of LIFE programs, the ith database predicate is denoted by dbp_i.

The design makes three assumptions.

- The subtype relations defined on the symbols appearing in the *dbp*'s is fixed.
- The storage order of tuples in the database relations is not relevant. This stems from the fact that order of *dbp*'s is not important in LIFE for unification. This assumption is needed to improve performance using different optimization techniques on the database predicates.
- The functor name denoting *dbp* (facts) is not allowed to appear on the left hand side of a rule. This does not in any way limit the computational power of LIFE.

The interface can be divided into three layers consisting of a LIFE meta-programming part for the *program analyzer* PA, *meta-interpreter* MI, and a *communication module* CM for physical communication with the database.

3.2.1 Program Analyzer

The *program analyzer* takes as input pure LIFE code and rewrites it into a modified LIFE code, containing the original rules and meta-rules of the Meta-interpreter in place of the fact base. It analyzes the LIFE code, identifying the facts that are to be stored in the database and partitions them into non-overlapping segments (*qualified segments*) based on the type-hierarchy. Each such partition of the fact base is stored as a separate relation in the database and replaced in the modified LIFE code by the meta-rules. The meta-rules include a database-schema (*qualifier*) for each relation stored in the database for interaction between LIFE system and database.

Statistical profiles of each relation are also computed and stored in the appropriate qualifier. Such profiles are necessary: for instance, the cardinality of each relation can be used to infer that all *dbp*'s from the relation are in main-memory requiring no further interaction with the database for this relation.

The user can then start a LIFE session by executing the compiled programs.

3.2.2 Meta-interpreter

The Meta-interpreter provides a tight coupling, retrieving *dbp*'s from the external database on demand. It include rules for matching the database predicates (LIFE predicates ,that are stored in the external database) with the main-memory resident facts and facts in the relational database. The MI is activated whenever an attempt is made to resolve a particular *dbp* g, called the *goal.*

It first checks the core-resident database facts, unifying the first *fact* with the goal *g*. On successive backtracking, the inference engine examines the next possible *fact* in the internal database for unification. If no further facts in main-memory are available, the MI activates the CM for retrieving unifiable facts from the external database.

Crossing the boundary from LIFE to database is an expensive operation. To achieve the goal of efficiency, the MI keeps track of the past interaction with the database, reducing the number of calls to the database, as it never repeats any query, subsumed by past queries. In this process it also ensures that no data item is retrieved twice.

The basic algorithm of the meta-interpreter can be informally described as consisting of following stages:

1. *Match main-memory facts*: For a goal *g*, we examine the facts in main-memory that can unify with g. If the internal database is exhausted we go to the next stage.

2. *Examine the restrictor set*: We examine a new data-definition (qualifier) whose name does not appear in the restrictor set. The restrictor set contains names of relations, for which all the tuples have been retrieved.

3. *Generate a new candidate*: The qualifier contains meta-information about the contents of its relation (qualified segment). A candidate C is generated if the *qualified segment* contains unifiable *dbp*'s with the goal *g*; otherwise we go back to step 2 to examine the next qualifier for g. The candidate C contains all the information for selective retrieval of *dbp*'s.

4. *Check for subsumption*: This stage checks if the candidate for the current goal is subsumed by candidates generated by previous queries. If so, the *dbp*'s from the relation have been fetched for this goal and we go back to step 2 to process the next qualifier for g.

5. *Generate negative candidates*: The answer needed by the current query can overlap with previous queries. Negative candidates are generated so that proper SQL query can be generated, which only loads the *dbp*'s once.

6. *Load new* dbp's *and resume unification*: The last stage loads the *dbp*'s from the external database. The newly asserted *dbp*'s are unified with the *goal* g, and if further data items are needed we go back to step 2.

Two main issues in the design of MI are:

- *Problem1*: activating the *CM* to retrieving new facts from the external database, whenever the active memory is exhausted of matching facts.

- *Problem2*: when the loading process is done, the MI should enable the continuation of unification of the current dbp_i with the newly asserted facts.

The LIFE system maintains predicates with the same functor name in a linked list. Any retrieved facts from the external database have to be added to this list. An active instance of a database predicate may need to access the database several times. Multiple occurrences of the same dbp_i can be active at the same time accessing the database. This could lead to potential problems and inconsistencies [16].

Problem 1 and Solution

Suppose we have a set of facts named dbp_i in our original program that needs to be stored in the external database. These are asserted in the external database, the transformed LIFE code containing meta rules of the same functor dbp_i for data retrieval from the database.

The MI provides tight coupling loading dbp_i from the external database on demand. Any new database facts dbp_i retrieved from the EDB are asserted at the end of chain of facts dbp_i. In the beginning, the database predicates with the same functor consists of meta-rules of the form:

(1) *persistent(qpointer)?* [1]

(2a) F : dbp_i :-
have_to_set_dbm,!,
set_dbm(dbp ⇒ F).

(2b) F : dbp_i :- *load_facts(qpointer, dbp ⇒ F), fail.*

(2c) dbp_i :- *fail.*

The rule (2a) stays at the start of the chain of dbp_i. Its purpose will be explained later. Rule (2b) called *dbm* rule always stays at the end of dbp_i chain. It activates the *communication module* CM and inserts the retrieved facts at the end of dbp_i chain. The *dbm* rule (2b) is retracted and reasserted at the end of the chain. Since the *dbm* rule is always placed at the end of the chain, this rule is only considered when all instances of the facts in the active memory have been attempted to resolve the current goal dbp_i. The forced failure of this rule, at the end of database retrieval, enables the inference engine to continue unification of current goal with the newly asserted facts.

At the end of loading process, a *failure rule* (2c) needs to be asserted at the end of the predicate chain to ensure the correctness of the backtracking

[1]The predicate *persistent* is a built-in LIFE predicate, which declares **qpointer** to be a persistent variable. The notion of persistent variables and terms are explained in section 1.3.2.

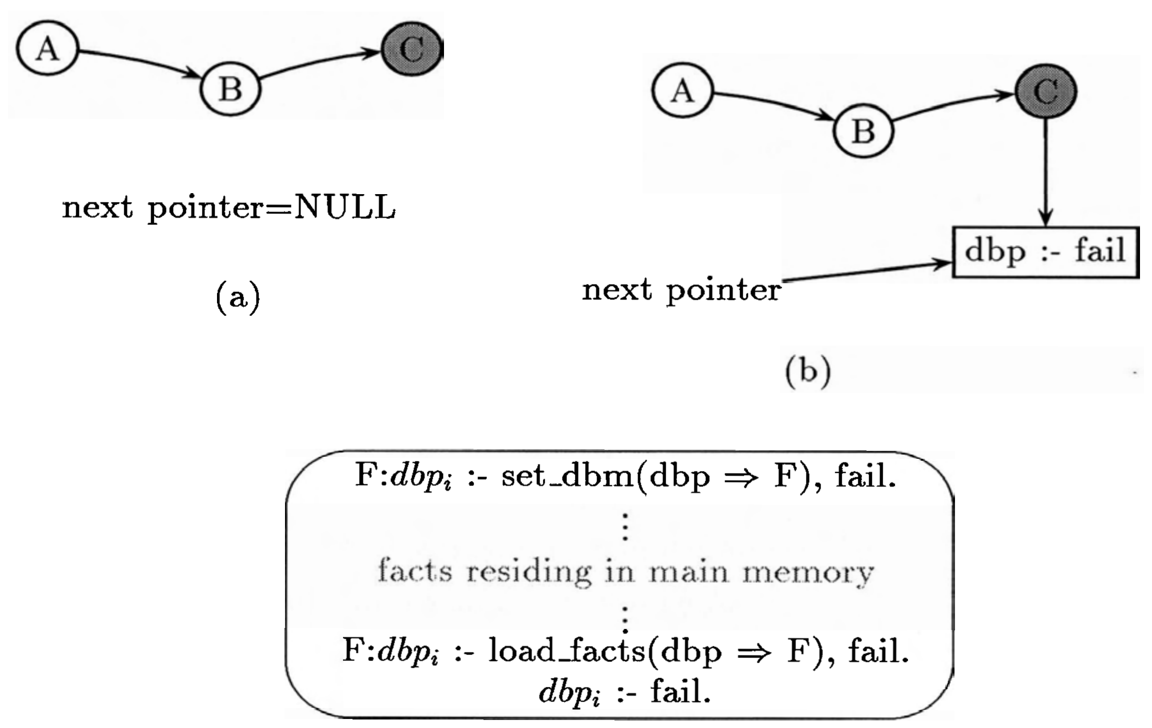
A
B
C
next pointer=NULL
(a)
A
B
C
dbp :- fail
next pointer
(b)
F:dbp_i :- set_dbm(dbp ⇒ F), fail.
facts residing in main memory
F:dbp_i :- load_facts(dbp ⇒ F), fail.
dbp_i :- fail.

Figure 3.2: Failure rule for Unification of Database Facts

mechanism of the LIFE interpreter. The LIFE system maintains predicates with the same functor name in a linked list (Figure 3.2 shows three clauses of the same functor name in a chain). The unification process keeps a pointer to the current unified element, and a pointer to the next element in the list for optimization purpose.

This presents a problem when the loading process is done and the inference engine needs to continue with the unification of the current *dbp*$_i$ with the newly asserted facts. Suppose the current *dbp*$_i$ is unified with the last element in its chain (Node *C* in the chain of clause *dbp*$_i$ in figure 3.2) (which is the *dbm* rule for it), the *next* pointer for it points to nil. The activation of *dbm* rule could retrieve facts from the EDB and assert the retrieved facts at the end of the predicate chain. On backtracking from the *dbm* rule, the inference engine is unaware of new facts retrieved from the EDB, as its next pointer is not reset to point to the newly retrieved facts.

This problem is avoided by asserting a *failure rule* (dbp_i :- fail) whenever the *dbm rule* is asserted at the end of the predicate chain. This ensures that the *next pointer* points to the *failure rule* (figure 3.2) rather than being *nil* for any activation of a database predicate. The retrieved facts are appended after the *failure rule*, and on backtracking, the inference engine can access these facts.

```
(3)  have_to_set_dbm :-
            (access_main_memory_facts,!,
            retract(access_main_memory_facts), fail;
            assert(access_main_memory_facts)).

(4)  set_dbm(dbp => Dbp) :-
            Old = qpointer,
            data_definition(dbp => Dbp, qpointer),
            resolve_dbp(Old, dbp => Dbp).

(5a) resolve_dbp(dbp => Dbp) :- Dbp.

(5b) resolve_dbp(Old) :- qpointer <<- Old, fail.
```

Problem 2 and Solution

A second problem arises when multiple instances of the same database predicate dbp_i are active at the same time. The multiple *data definitions* for each dbp_i are maintained as a list of elements in LIFE. If we access a single relation at a time, each activation of dbp_i needs to maintain a pointer *qpointer* to the next *data definition* it is going to access when the *dbm* rule is fired for it.

Whenever a new instance of dbp_i is activated, rule (2a) saves the *qpointer*, and sets *qpointer* to the start of data-definition list for the current instance. It resets *qpointer* to the value stored in Old, when the current (i.e., new) instance is to backtrack.

```
(6a) load_facts([Qual | Tail], dbp => Dbp) :-
            not_in(restrictor(Qual)),
            candidate(C, dbp => Dbp, qualifier => Qual),
            retrieve_facts(C),!,
            qpointer <<- Tail,
            retract_dbm,
            assert_dbm.

(6b) load_facts([_ | Tail], dbp => Dbp) :-
            load_facts(Tail).
```

The predicate (6a) is activated by the *dbm rule* when a match for core-memory resident *fact* fails. It looks up a new *data definition* for the database predicate dbp_i. A list of restricted_set of relations is maintained, whose entire set of tuples has been retrieved and asserted in the main-memory. If the relation name for the data-definition is found in this set no further interaction with the database is needed for this relation. The clause fails, and backtracks to examine the next data-definition (*qualifier*) for dbp_i.

The *qualifier* contains meta-information about the contents of its relation (qualified segment). A candidate C is generated if the relation (*qualified segment*) contains potential for containing clauses which can unify with the current database predicate dbp_i, otherwise we backtrack to examine the next qualifier for dbp_i. The candidate C provides a selection condition retrieving subset of tuples from the relation that can unify with dbp_i.

```
(7a) retrieve_facts(C) :-  subsumed_candidate(C),
                           !, fail.
```

The next phase consists of checking whether the candidate C is subsumed by candidates generated by previous queries. If so the facts from the relation have been fetched by a more general query, and the relation contains no new facts which need to be fetched, that can unify with dbp_i. The clause fails and goes to examine the next *data definition*.

(7b) *retrieve_facts(C) :- assert_candidate(C), overlap_candidates(C, NC), read_facts(C, NC)*

The last phase accesses the database, asserting the retrieved facts as well as maintaining meta-information about the current query. The answers needed by the current query can overlap with previous queries. The *overlap_candidates* predicate identifies any such overlap so that proper SQL query can be generated, which only loads the facts once. The candidate generated for the current query is cached as well to avoid sending the same or *subsumed* database query to the external database in future.. The generated candidates are compacted and stored in an efficient manner. Compaction is done as follows: It first checks if the current candidate subsumes any previous candidate. If that is the case then the past candidate list is pruned by removing the subsumed candidate, as the current candidate is more general. A second compaction is done, removing the candidates generated on a relation, if all the tuples from the relation are in main memory. The names of such relations are kept in a *restrictor* set.

When the loading process is done, the MI enables the continuation of unification of the current dbp_i with the newly asserted facts.

3.2.3 Communication Module

The *communication module* (CM) establishes the physical communication between LIFE and the database. The communication module is written in C++ providing new predicates for interacting with the database. Both PA and MI can call the CM. From the information passed on by these two modules, it constructs an appropriate SQL query, executes the query, and converts the data from one format to the other. The CM is built in an modular fashion to enhance portability to other database systems. It contains two modules, the *formatter* and the *DBMS interface library.* The former translates queries and data between the two different representation, while the latter is used for communication with the DBMS, submitting queries and collecting answers. The behavior of the two modules is encapsulated in a C++ abstract base classes, and appropriate derived classes can be provided to interact with a different database system.

3.3 Storing and Retrieving LIFE Facts

In this section we present the theory [1] developed to store and retrieve LIFE *facts* in an external database.

The complex data-structure of *facts* cannot be straightaway translated to a relational database. Therefore an intermediate representation is provided that maps LIFE *facts* to relational *tuples* and vice-versa. Meta-information (in the form of *candidates*) on the relations, provides selective retrieval of tuples, which can unify with the current LIFE goal. The queries and their answers are cached, so that the *facts* are retrieved only once.

The theory proposed in [1] is for *variable free facts* only. Moreover it restricts facts, for which the symbols occurring in these facts are involved in single inheritance only. We generalize the technique to store facts containing variables and type symbols involved in multiple-inheritance. The solution presented for the data retrieval in the paper [1] is extended, as it cannot handle variables in a LIFE query. We present a solution for it and further optimize the data retrieval by exploiting the type information stored in the *qualifier*.

3.3.1 Sample Program

We show a sample program, the facts in which are to be stored in an external database.

Example 3.1 *This example shows a small sample LIFE program, consisting of:*

%%% Facts (Unit clauses):

V_1: *vehicle_db(owner ⇒ adams, item ⇒ car(make ⇒ nissan, model ⇒ '280zz')).*

V_2: *vehicle_db(owner ⇒ viraj, item ⇒ car(make ⇒ ford, model ⇒ aerostar)).*

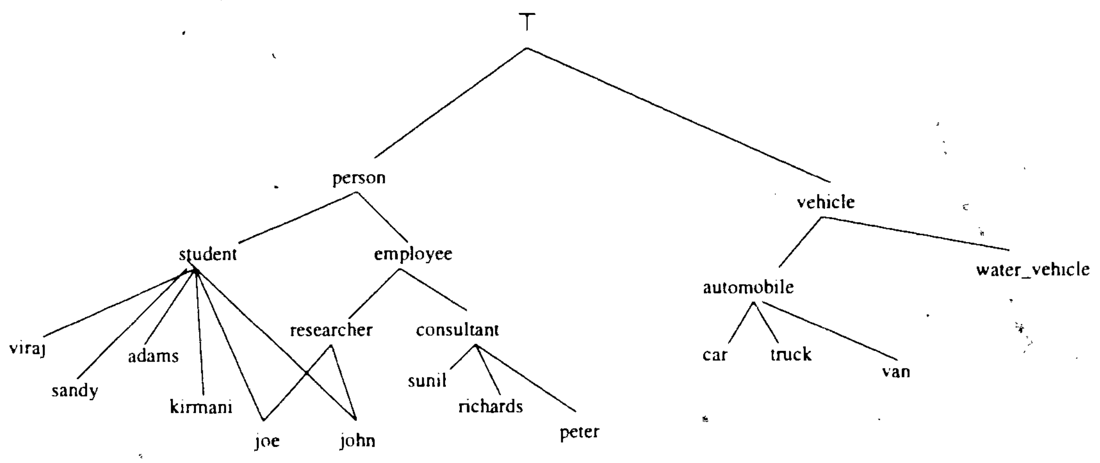

Figure 3.3: Type hierarchy for example 3.1

V_3: *vehicle_db(owner ⇒ sandy, item ⇒ car(make ⇒ hero, model ⇒ jet)).*

V_4: *vehicle_db(owner ⇒ joe, item ⇒ van(make ⇒ panther, model ⇒ ghia)).*

V_5: *vehicle_db(owner ⇒ john, item ⇒ car(make ⇒ maruti, model ⇒ xlr)).*

V_6: *vehicle_db(owner ⇒ john, item ⇒ van(make ⇒ panther, model ⇒ cdx)).*

%%% Type definitions

person	:= { *student; employee* }	*% a student as well an employee is a person.*
student	:= { *viraj; adams; joe; john* }	*% viraj, adams, joe and john are students.*
student	:= { *sandy; kirmani* }	*% sandy, kirmani are also students.*
researcher	:= { *joe; john* }	*% joe and john are researchers as well*
consultant	:= { *sunil; peter; richards* }	*% sunil; peter; richards work as consultants*
employee	:= { *researcher; consultant* }	*% an employee could be a researcher or a % consultant.*
vehicle	:= { *automobile; water_vehicle* }	*% a vehicle could be a automobile or a % water_vehicle.*
automobile	:= { *car; van; truck* }	*% an automobile could be a car, van, or a % truck.*

Figure 3.3 shows the type hierarchy for this program representing partial order on the type symbols { person, student, employee, researcher, consultant, viraj, adams, joe, john, sandy, kirmani, sunil, richards, peter, vehicle, automobile, water_vehicle, car, van, truck }. It is assumed that the type hierarchy is fixed.

3.3.2 Representation of Facts in a Relational Model

Intermediate representation

A LIFE *fact* is a complex object containing cycles and structure sharing by means of coreference constraints (using variables, explained in section 1.1.1). In [1] an equivalent mathematical construct for the (variable free) complex structured facts is shown, which can then be represented in a relation. We call this equivalent mathematical construct as the flattened ψ-term.

The *facts* in LIFE are based on the ψ-term data-structure of LIFE, consisting of *type symbols* and *labels*. Example 3.2 shows a ψ-term and the flattened ψ-term T_p for it.

Example 3.2 *For the ψ-term:*

```
vehicle_db ( owner ⇒ john,
             item  ⇒ car ( make  ⇒ ford,
                           model ⇒ aerostar
                         )
           ).
```

The vehicle *ψ-term can be represented as a relational tuple T_p = {ε: vehicle_db, owner: john, item: car, item·make: ford, item·model : aerostar},*

Qualified segments

Each flattened ψ-term can be stored as a relation in the database, but this leads to a large number of relations in the database. We can exploit the sub-type information present in the type hierarchy to club items together to be stored as one relation. All the items grouped together have the same structure. Given a flattened ψ-term we replace each entry by its parent, the resulting structure is called a *qualifier*. The *facts* having the same *qualifiers* are stored as tuples of a relation. All the facts stored under one relation are referred to as a *qualified segment*. Associated with each *qualified segment* Q is a *qualifier* denoted by qual(Q).

Example 3.3 *For the type hierarchy in Fig 3.3 the type symbols parents are:*
$par(person) = par(vehicle) = \{\top\}$
$par(student) = par(employee) = \{person\}$
$par(researcher) = par(consultant) = \{employee\}$
$par(viraj) = par(adams) = par(sandy) = par(kirmani) = \{student\}$
$par(john) = par(joe) = \{student,\ researcher\}$
$par(sunil) = par(richards) = par(peter) = \{consultant\}$
$par(automobile) = par(water\_vehicle) = \{vehicle\}$
$par(car) = par(van) = par(truck) = \{automobile\}$

From the parent definitions, the qualifier for the predicate example 3.2 is then constructed as:

{ ϵ:$\top$, *owner:[student, researcher], item:automobile, item·make:*$\top$, *item·model:*$\top$ }.

Recall from section 1.1.2 that $\top$ is a special symbol in LIFE which subsumes every other symbol. If a symbol's parent is not explicitly stated it defaults to $\top$.

The facts are grouped together in a *qualified segment* Q, where all facts in the segment have the same *qualifier*. One can easily see that the structure of facts in the *qualified segment* is the same, since for any two ψ-terms (f, f') in Q, we have $qual(f) = qual(f')$.

The flattened representation of facts in a *qualified segment* Q are stored in a relation as tuples.

Example 3.4 *Example 3.1 shows a LIFE program containing six facts. The type hierarchy for the symbols occurring in these facts is shown in figure 3.3*

The parents of the type symbols occurring in the first three facts are the same. So is the case for the remaining three facts. These facts then can be represented in the two qualifier segments:

Q_1=[V_1: {name: adams, item: car, item·make: nissan, item·model: '280zz'},
V_2: {name: viraj, item: car, item·make: ford, item·model: aerostar},
V_3: {name: sandy, item: car, item·make: hero, item·model: jet}].

Q_2=[V_4: {name: joe, item: van, item·make: panther, item·model: ghia},
V_5: {name: john, item: car, item·make: maruti item·model: xlr}
V_6: {name: john, item: van, item·make: panther item·model: cdx}].

The qualifiers being

qual(Q_1) = {name: student, item: automobile, item·make: $\top$, item·model: $\top$}

qual(Q_2) = { name: [student, researcher], item: automobile, item·make: $\top$, item·model: $\top$}

The qualifying segments Q_1, Q_2 can be stored as two relations (R_1, R_2) in the database. The mapping between LIFE *facts* and *tuples* in the relation is shown in the next example, using the concept of *data definition.*

On a closer inspection of the relation R_1 in table 3.1, we find redundancy in the representation of the vehicle database. In particular the *item* occurrence (column c2) has the same symbol *car* occurring in all their tuples. Such symbols can be best represented in the qualifier itself reducing the size of the table. The relational table R_1 is compacted by removing the column c2 and the type symbol stored in the modified qualifier called *data definition.*

For each qualified segment Q, a *data definition* is constructed, which handles the transformation of data between a LIFE ψ-term and relational tuple. The next example illustrates how this can be done by means of coreference constraints in LIFE.

Table 3.1: A vehicle database

(R_1)

	c1	*c2*	*c3*	*c4*
V_1	adams	car	nissan	280zz
V_2	viraj	car	ford	aerostar
V_3	sandy	car	hero	jet

(R_2)

	c1	*c2*	*c3*	*c4*
V_4	joe	van	panther	ghia
V_5	john	car	maruti	xlr
V_6	john	van	panther	cdx

Example 3.5 *For the example program 3.1, the* data definitions *for the vehicle database qualifiers 3.4 can be represented in LIFE as:*

```
D1 =
data_definition(
  structure ⇒ vehicle_db( name ⇒ A, item ⇒ car(make ⇒ C, model ⇒ D)),
  tuple     ⇒ R1( c1 ⇒ A, c3 ⇒ C, c4 ⇒ D),
  qualifier ⇒ R1( c1 ⇒ student, c3 ⇒ ⊤, c4 ⇒ ⊤) ).


D2 =
data_definition(
  structure ⇒ vehicle_db( name ⇒ A, item ⇒ B(make ⇒ C, model ⇒ D)),
  tuple     ⇒ R2( c1 ⇒ A, c2 ⇒ B, c3 ⇒ C, c4 ⇒ D),
  qualifier ⇒ R2( c1 ⇒ [student, researcher], c2 ⇒ automobile, c3 ⇒ ⊤, c4 ⇒ ⊤) ).
```

The subterm denoted by the *structure* label in the data definition provides the *data-structure* of the ψ-terms stored in the relational database. The coreference constraint between the subterms denoted by *structure* and *tuple* labels provides the translation between the data-representation of LIFE and relational database.

Column c2 in the relational table R_1 is removed and the symbol *car* is represented in the *data definition* D_1 (in the subterm denoted by the label *structure*) itself.

Assume a *goal* g = vehicle_db(owner ⇒ researcher, item ⇒ vehicle(make ⇒ panther)). We unify with the subterm denoted by the structure label in the data-definitions, constructing a simple query (subterm denoted by tuple label). We call this the *SQL goal* for g. For this *goal* g, the SQL goals generated are: R_1(c1 ⇒ researcher, c3 ⇒ panther, c4 ⇒ ⊤), R_2(c1 ⇒ researcher, c2 ⇒ vehicle, c3 ⇒ panther, c4 ⇒ ⊤).

Similarly, any tuple retrieved by the CM of the interface, is of the form of subterm denoted by the tuple label, and on unification of the retrieved tuple with this subterm, it is translated to LIFE format (subterm denoted by structure label).

3.3.3 Data Retrieval

The LIFE database interface accepts arbitrarily *complex goals* and returns all potentially unifiable facts with the *goal* from the external database. Whenever a *goal* g cannot be resolved by the facts in the internal LIFE database, the interpreter needs to fetch the facts from the database. This is done using the concept of candidates as explained below.

Candidate

We can use a brute-force technique, by retrieving the facts one-by-one from the database, until we get a *fact* which unifies with the *goal.* However, since this is highly inefficient, optimization is done by retrieving only a subset of facts from a *qualified segment* Q_i, which can unify with the current *goal* g.

In order to do so, a *candidate* C is constructed for each relation: based on the *SQL goal* constructed for the *goal* g and the *qualifier* defined on the relations.

The *candidate* C has identical structure as that of the SQL goal and the qualifier, consisting of $\top$ symbols and immediate subtypes of symbols in the qualifier (which are also the symbols that appear in the qualified segment).

If the symbol [2] in the qualifier for label c_i is a subtype of the symbol referenced by the label c_i in SQL goal, the candidate consists of $\top$ symbol for the label c_i. The $\top$ symbols in C are wild card entries [3], since the goal symbol subsumes (and thus unifies with) the symbols occurring at this label for all the facts in the qualified segment.

If the SQL goal symbol does not subsume the qualifier symbol, the symbol referenced by the label c_i in the candidate C contains a non-empty list of symbols (immediate subtypes [4] of the symbol referenced by the label c_i in the qualifier) which can unify with the corresponding symbol in the SQL goal. Since the immediate subtypes of symbols in the qualifier are same as symbols that appear in the facts in the qualified segment, the non-top symbols in C thus provide for selective retrieval of tuples for the relation that can help resolve the *goal* g.

Example 3.6 *Given the* vehicle database *in example 3.4, consider the following query: g_1: vehicle_db(owner $\Rightarrow$ researcher, item $\Rightarrow$ van).*

From the data-definitions (D_1, D_2) shown in example 3.5 for vehicle_db, only D_2 will generate a candidate.

The candidate constructed for D_2 is: C = candidate(R_2(c1 $\Rightarrow \top$, c2 $\Rightarrow$ [van], c3 $\Rightarrow \top$, c4 $\Rightarrow \top$)). The $\top$ in C is a wild card argument (indicating that there is no selection condition for c_i in the relation for retrieval of tuples as all of the symbols for c_i in the relation will unify with the goal *symbol) and non-top symbols are the selection arguments for tuples in relation R_2. The selection*

[2]if it is a list of symbols, any one of the symbols in the list

[3]indicating that there is no selection condition for c_i in the relation for retrieval of tuples

[4]For multiple inheritance it is a subset of immediate subtypes of list of symbols (S) referenced by the label c_i in the qualifier, whose parents are the same as the list of symbols S.

condition for C is S(C) = (c2=[van])(as the relation contain immediate subtypes of the type automobile *for c2). we select the tuples with a simple SQL query:*

select c1, c2, c3, c4
from R_2
where S(C)

For the sample query, we get the tuples in table

	c1	*c2*	*c3*	*c4*
V_4	joe	van	panther	ghia
V_6	john	van	panther	cdx

(R_2)

which is then transformed to the facts

V_4: vehicle_db(owner ⇒ joe, item ⇒ van(make ⇒ panther, model ⇒ ghia)),

V_6: vehicle_db(owner ⇒ john, item ⇒ van(make ⇒ panther, model ⇒ cdx)).

Caching queries and answers

The backtracking mechanism of LIFE may result in sending the same query to the database.

Example 3.7 *For example, consider a LIFE program that has to evaluate the following clauses:*
$pred_i(A)$:- . . . , $pred_j(A, B)$, $dbp_i(B, C)$, ...
$pred_j(b, a)$.
$pred_j(c, a)$.
$pred_i(A)$?

It is clear that the database predicate "$dbp_i(a, C)$" will be executed twice, once when $pred_j$ is resolved with its first ground clause "$pred_j(b, a)$" and a second time when on backtracking $pred_j$ matches its second ground clause "$pred_j(c, a)$". In order to minimize the interaction with database, and avoid repeating the same query, the candidates for the queries and their answers are cached in the active memory. This technique known as caching queries is described in [14] for Prolog, and is generalized for ψ-terms in LIFE.

In fact we need to avoid sending a subsumed query to the database again, i.e., we check whether the candidate for the query is going to result in fetching the subset of facts which have been retrieved by a previously generated candidate. Subsumption of queries can be easily checked in LIFE, as it corresponds to checking the sub-type relations on the type symbols in the candidates generated for the queries for the same relation. Recall that from the definition of a candidate the attribute values of a candidate is either $\top$ or a list of symbols other than $\top$. A candidate C_i is subsumed by a candidate C_j (both candidates are for the same relation), if for each feature in C_j, one of the following is true:

- The feature value is $\top$, if not,
- the corresponding attribute value in candidate C_i is not $\top$, and the list of symbols in C_i for this feature is a subset of the list of symbols in C_j from the same feature.

Example 3.8 *Assume the goal g_1 = vehicle_db(owner $\Rightarrow$ researcher, item $\Rightarrow$ van), and the qualified segments in example 3.4. From the data-definitions (D_1, D_2) for vehicle_db, we construct the candidate C_1: candidate(R_2(c1 $\Rightarrow$ $\top$, c2 $\Rightarrow$ [van], c3 $\Rightarrow$ $\top$, c4 $\Rightarrow$ $\top$), which returns the fact $\{V_4, V_6\}$.*

For goal g_2 = vehicle_db(owner $\Rightarrow$ john, item $\Rightarrow$ van), we construct candidate C_2 = candidate(R_2(c1 $\Rightarrow$ [john], c2 $\Rightarrow$ [van], c3 $\Rightarrow$ $\top$, c4 $\Rightarrow$ $\top$).

C_1 subsumes C_2, the symbol john *in C_2, providing a further selective condition on tuples in R_2. The facts going to be fetched by C_2 will be a subset of facts already retrieved by C_1 and need not be loaded again.*

Negative candidates

A query need not be subsumed by previous queries, but there could be an overlap of facts that needs to be retrieved by the current query and facts already loaded by the previous queries. Let the set of candidates S_c represent parts of a relation R_i already loaded. C is the candidate constructed for the current query on the relation R_i, and is not subsumed by any of the candidates C_i in S_c.

We retrieve the facts for the candidate C, but exclude the facts already loaded by the previous set of candidates S_c. We need to consider a candidate C_i from S_c only if it has loaded facts which can overlap with the facts needed by C.

As mentioned earlier the arguments of a candidate are either $\top$ symbol or a list of symbols without the top. The intersection (overlap) of the set of facts loaded by C and C_i is non-empty if for all corresponding arguments in the two candidates, either of the argument value in C, C_i is $\top$ or the intersection of the two lists is non-empty.

Example 3.9 *Suppose we have already loaded some facts from relation R_2 using the candidates, $((C_1$ = candidate$(R_2(c1 \Rightarrow [john], c2 \Rightarrow [car], c3 \Rightarrow \top, c4 \Rightarrow \top), C_2$ = candidate$(R_2(c1 \Rightarrow [john], c2 \Rightarrow [van], c3 \Rightarrow \top, c4 \Rightarrow \top))$. The facts retrieved are $\{V_5\}$, and $\{V_6\}$.*

To retrieve all the facts for the goal g = vehicle_db(owner $\Rightarrow$ researcher, item $\Rightarrow$ car), we construct the candidate C = candidate$(R_2(c1 \Rightarrow \top, c2 \Rightarrow car, c3 \Rightarrow \top, c4 \Rightarrow \top)$.

The intersection of C and C_1 is non-empty, as the corresponding symbols in them are equal, or one of them is $\top$, while C and C_2 do not load any common facts.

We select the tuples with an SQL query:

select c1, c2, c3, c4

from R_2
where $S(C)$ *and not* $S(C_1)$.

$S(C)$ = *(c2=car) is the selection condition for* C, *and* $S(C_1)$ = *(c1 = john and c2=car) is the selection condition for* C_1.

Candidate optimization

To minimize the interaction with the database, we assert the retrieved facts in the internal database of LIFE, and also cache the candidates generated. The storage of candidates is expensive. We reduce the number of candidates, whenever a new candidate is added to the set of candidates, by removing the candidates which are subsumed by the new candidate.

3.3.4 Improvisations

Variables in ψ-terms

Variables in ψ-terms are used to denote *coreference constraints* (structuring sharing) between subterms in it. In [1] the theory presented is for *variable free* facts. We feel that the constraint that the term should contain no variables is too restrictive for database applications (like NLP). We will demonstrate with an example how to handle such constraints and store the facts containing such constraints in the relational database.

Example 3.10 *We consider an example of a parent database containing the names of the parent and the child, and the addresses where they live. If the child lives with his parent this can be reflected by means of coreference constraint on the addresses of the parent and child. The type hierarchy for this example is shown in figure 3.3.*

%%% Facts:
P_1: *parent(son* $\Rightarrow$ *viraj(address* $\Rightarrow$ *X:address_string*$_1$*),*
father $\Rightarrow$ *richards(address* $\Rightarrow$ *X)).*

P_2: *parent(son $\Rightarrow$ sandy(address $\Rightarrow$ X:address_string$_2$),*
father $\Rightarrow$ sunil(address $\Rightarrow$ X)).

P_3: *parent(son $\Rightarrow$ adams(address $\Rightarrow$ address_string$_3$),*
father $\Rightarrow$ peter(address $\Rightarrow$ address_string$_4$)).

P_4: *parent(son $\Rightarrow$ kirmani(address $\Rightarrow$ address_string$_5$),*
father $\Rightarrow$ richards(address $\Rightarrow$ address_string$_6$)).

The constraint that the addresses of parent and child in (P_1, P_2) are same can be represented in the data-definition itself.

The two facts (P_1, P_2) can be grouped together in the same qualified segment Q_3, and the data_definition for it is:
D_3 =
data_definition(
structure $\Rightarrow$ parent(son $\Rightarrow$ A(address $\Rightarrow$ X), father $\Rightarrow$ B(address $\Rightarrow$ X))
tuple $\Rightarrow$ R_3(c1 $\Rightarrow$ A, c2 $\Rightarrow$ B, c3 $\Rightarrow$ X),
qualifier $\Rightarrow$ R_3(c1 $\Rightarrow$ student, c2 $\Rightarrow$ consultant, c3 $\Rightarrow$ string)).

The facts (P_3 and P_4) have the same qualifier as the two facts (P_1 and P_2), but do not belong to the segment Q_3, since the coreference constraint in them does not match with P_1 and P_2. We store P_3, P_4 in a separate qualified segment, the data_definition for Q_4 is:
D_4 =
data_definition(
structure $\Rightarrow$ parent(son $\Rightarrow$ A(address $\Rightarrow$ X), father $\Rightarrow$ B(address $\Rightarrow$ Y))
tuple $\Rightarrow$ vehicle_db(c1 $\Rightarrow$ A, c2 $\Rightarrow$ B, c3 $\Rightarrow$ X, c4 $\Rightarrow$ Y),
qualifier $\Rightarrow$ R_4(c1 $\Rightarrow$ student, c2 $\Rightarrow$ consultant, c3 $\Rightarrow$ string, c4 $\Rightarrow$ string)).

Variables in goals

The retrieval algorithm presented in [1] retrieves more facts than is needed. In this section we augment the algorithm so that it handles variables in *goal* in a more efficient manner.

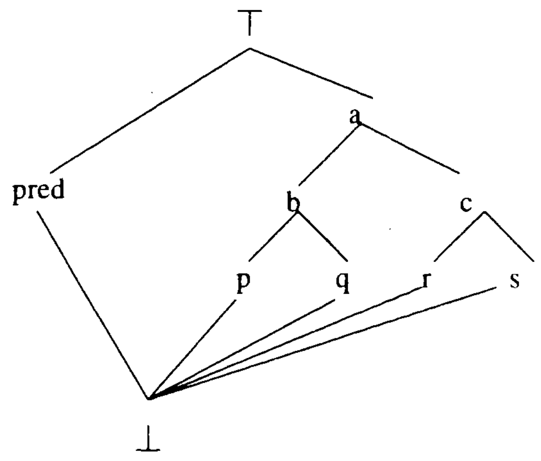

Figure 3.4: Type hierarchy for example 3.11

Example 3.11 *For example, consider the two facts: {pred(p,r), pred(q,s)}. The type hierarchy for it is shown in figure 3.4. The* data definition *constructed for the two facts is data_definition(structure ⇒ pred(A,B), tuple ⇒ Rel(c1 ⇒ A, c2 ⇒ B), qualifier ⇒ Rel(c1 ⇒ b, c2 ⇒ c)).*

Assume a query pred(X:a, X), the SQL goal for it is Rel(c1 ⇒ X:a, c2 ⇒ X), and the candidate constructed for it is candidate(c1 ⇒ $\top$, c2 ⇒ $\top$).

This will fetch the set of two facts {pred(p,r), pred(q,s)}. The result is incorrect, as neither fact can unify with the goal.

An observation we make here is that if there is any fact in a qualified segment that can unify with a goal, then the qualifier for the segment will have to unify with the goal. This provides a coarse filter to see if the qualifier needs to be further processed to generate a candidate for the goal. The same mechanism provides a partial solution for goals containing variables. For the above query, the candidate is not generated at all, as the qualifier fails to unify with the SQL goal. This will not result in any database access for the goal now.

3.4 Garbage Collection

LIFE memory becomes polluted when the retrieved database facts are cached in memory. If the database size is large, LIFE memory may become full when a large chunk of database facts have been retrieved preventing further computation. When this happens the cached database facts are automatically evicted to free up space. An LRU policy is used, where the least recently used fact name is retracted from the main memory. The number of facts to evict is a percentage of the retrieved database facts set by the user.

3.5 Conclusion

We have described the design and implementation of a database interface for LIFE. This interface provides for storage of complex facts as flat tuples in a relational database. We have extended the approach provided in [1] to store facts containing coreference constraints and multiply inherited types. The theory [1] could not handle variables in queries very well. We provided a solution to handle the case where variables occur in a LIFE goal.

The tightly coupled approach provides an efficient cache mechanism which enables applications to retrieve a smaller working data set in its main memory. The interface insulates the user from database operations. The compiled programs can be used directly by other user programs, requiring no additional support. The only limitations on the LIFE programs is not to contain *assert* or *retract* operations on the database predicates.

Chapter 4

Persistent Programming

4.1 Introduction

Over the past ten years much research effort has been directed at attempts to build persistent programming languages [21, 27, 33, 35, 60], incorporating database functionality into their programming models. The basic idea behind such systems is the concept of *orthogonal persistence* [21, 22]. *Persistence* is defined as the length of time, for which the data lives and is usable. The two basic principles behind orthogonal persistence are:

- any object may exist for as long, or as short, a period as the object is required
- an object may be manipulated in an uniform manner regardless of the length of time it persists.

In this sense persistent systems provide uniform abstraction over the storage.

Data in LIFE like other conventional programming languages, is short term (exist for a program lifetime). Storage for long lived objects is usually provided by a file system or a database interface. This results in long lived data being treated in fundamentally different manner from short lived data. A main drawback is the need by the programmers for code that translates between disk-resident representation of data and the representation used during execution.

This mapping of data between long and short term storage results in penalty in terms of programmer design time and program run time. In a language with persistence, manipulation of data, whether they are short lived or meant to exist between program runs, is transparent to the user; there is no need for mapping from one representation to another. The advantages of persistent programming are:

- improving programming productivity and easing the programmer's task, when sharing of arbitrary data structures between invocation of programs and even between many different programs.
- avoiding ad hoc arrangements for storage of long-lived object and data conversion from one format to another.

In the next section we will address how to identify persistent objects and different techniques for loading them into virtual memory.

4.2 Issues in Persistent Languages

Long term data storage in persistent programming systems are generally provided by an object store, a conceptually infinite repository in which objects reside. The objects in such a repository cannot be directly addressed by the user programs. To manipulate these they must be moved from the object store into virtual memory in a manner that is transparent to the application programmer. While dealing with persistent objects an identifier (*Persistent identifier*-PID) by which the object is referred to in the store is likely to differ from an identifier (*Virtual identifier*-VID) by which the same object is addressed in the virtual memory. This is due to the fact that PID maybe arbitrarily long (typically 128 bits or more) in order to assign world-wide unique names and to deal with large number of objects compared to a virtual identifier which is typically 32 bits long.

4.2.1 Identifying Persistent Objects

A key design issue in supporting the existence of both temporary and long-lived objects, is identifying what objects should be persistent. Different systems employ different techniques [47, 43] and include

- A class type is marked explicitly persistent. All instances belonging to this class are then made persistent. The O++ [35] language extends C++ language using this approach.

- The transient or persistent nature of an object is decided when the object is created regardless of the type to which it belongs. An example of this approach is ObjectStore [46].

- A third approach is to make data-objects persistent if they can be reached from a set of specified roots. PS-algol [22] takes this approach.

4.2.2 Object Faults and Residency Checking

The attempt to use persistent objects that are not currently resident in virtual memory is termed as *object-fault* [47], involving identification of reference type (residency check) and the transfer of object contents.

The residency checking can be classified into two categories [42] (*edge marking and node marking*):

Edge marking: In the edge marking scheme, the object references are tagged as swizzled (notion of swizzling explained in 4.2.3) or not. A disadvantage of the *edge marking scheme* is that multiple copies of the references could be made, before loading the referenced objects. A costly mechanism is needed to identify copies of such references and swizzle them. Another way is to swizzle the reference, as soon as it is discovered. This may result in some unnecessary swizzling.

Node marking: In the node marking scheme, all references in a resident object to non-resident objects are changed to point to a proxy-object. The proxy-object contains a persistent pointer to locate the object on the disk. When the non-resident object is loaded the proxy-object persistent pointer is changed to the virtual memory pointer of the loaded object. Subsequent references incur the cost of an indirection. At some point, the proxy-objects are scanned to check if they are swizzled and bypassed, to remove the overhead of indirection.

Residency checks can be implemented explicitly in software, or performed implicitly in hardware using some kind of hardware trap for non-resident objects. If proxy-objects (node marking scheme) are used, they can be allocated in a protected memory region to trap the references to them and handle object-faulting.

For data retrieval, various techniques are employed. The next subsection gives an overview of the techniques in use, and their relative advantages and disadvantages.

4.2.3 Pointer Swizzling

The technique of changing a persistent identifier to a virtual memory address has become known as *pointer swizzling*, and can be approached in a number of ways:

1. Map the entire object store into virtual memory. This suffers from some of the same disadvantages as the use of file systems. This is only possible if persistent stores are small enough to be contained within the virtual memory. The advantage of this approach is that it eliminates the overhead for residency checking, to distinguish swizzled and unswizzled pointers.

2. An object's virtual address is of the same size as its persistent identifier. If the object's identifier is synonymous to its virtual memory address, no address translation is needed and the object contents are copied into the appropriate location in the virtual memory from the disk. However

if the needed region is already in use, swizzling is performed. Like above this limits the size of persistent stores to that of the virtual memory.

3. Translate the PID to a virtual address on each dereference via a lookup in a resident object-table. This approach does no pointer swizzling at all, but will involve a relatively expensive search of the resident object-table each time the object is accessed.

4. Perform the translation from PID-to-VID only once, by replacing the persistent pointer in the virtual address space with a main memory pointer to the object. This is done the first time an object is referenced by the process so that subsequent dereferencing incurs no translation penalty.

Among the options discussed above, the last option seems to provide an efficient large object store and is most often used to implement persistent object stores. *Pointer swizzling* [43, 34] in this case may be approached in various ways

Eager and Lazy swizzling

Pointer swizzling can be done at different times, swizzling at the earliest possible as in *pure eager swizzling* where all the references in main memory are swizzled in advance. In contrast in *pure lazy swizzling*, swizzling is performed when a pointer is being dereferenced. Pure lazy swizzling provides an incremental approach, using software checks to swizzle pointers on dereferencing by the application program at run-time. In between the two extremes we could have wide variety of swizzling techniques.

Pure eager swizzling has few a advantages that it avoids the overhead of testing the state of reference (swizzled or not swizzled), but requires the data set be identified before using it, or atleast bounding it. On the negative side it involves some computational expense of swizzling pointers that are never used. The data retrieved is less selective, requiring more memory than in pure lazy swizzling technique. Lazy swizzling on the other hand swizzles references on demand and avoids reference that is not read and therefore cannot be dereferenced by the application. It has the disadvantage of a software check to test whether a reference is a PID or a virtual identifier every time an object is accessed. Lazy swizzling can be done at various granularity levels: pointer-at-a-time, recursively swizzling pointers in an object upto a certain depth or swizzling all pointers in the page at once.

Hardware and Software based swizzling

Recently, a class of swizzling schemes [45] have been proposed that use virtual memory access protection technique to trigger the detection and transfer of non-resident persistent objects. The basic strategy is to allocate a page of virtual memory (access protected) to a non-resident object reference (virtual memory page maps to the page in the persistent store that contains the object). Accessing the object triggers a virtual memory trap, reading in the persistent page into the previously reserved virtual page. This approach avoids the overhead of residency checks incurred by software approaches, which makes the access to resident persistent objects as efficient as access to non-persistent objects.

In the next section we will use the concepts presented so far for the storage and retrieval of persistent data in LIFE in an object-store. Techniques like catching data, prefetching data, and clustering data in the database were studied for their impact on performance.

4.3 Persistency in LIFE

From the above discussion it is clear that there is a need for storing persistent ψ-terms in LIFE in a database. Recall that a persistent ψ-term unlike 'normal' ψ-term retain its value on backtracking, and can be viewed as a set of graphs with handles on certain nodes. The persistent terms are stored in an object store implemented on top of the file system of the operating system. As

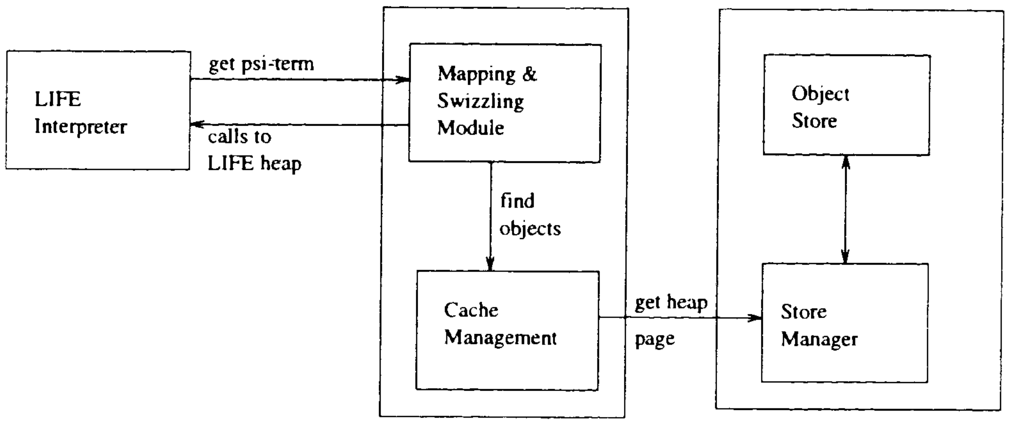

Figure 4.1: Persistent store architecture for LIFE.

mentioned previously this was done for portability reasons. The architecture of the system is depicted in Fig. 4.1. Portions of the LIFE interpreter have been rewritten so that when persistent data is encountered, special routines can be executed that will handle the persistent data.

An additional interface written in LIFE, compiles the original LIFE program into a modified LIFE program, storing the persistent terms in the object store. It provides transparent retrieval of the ψ-terms in the object store, and supports orthogonality, manipulating persistent and transient terms using the same compiled code. The interface supports programs written in pure LIFE, and the existence of a store under LIFE is made transparent to the user. The compiled code can also be linked to other applications written in LIFE, requiring no special support.

The following sections discuss each part of this system in detail.

4.3.1 Design Goals

- When designing a persistent system, a primary goal should be ease-of-use for the intended users. If allocation and manipulation of persistent objects is no different from manipulation of short-lived objects, the program will be easier to write. This also allows for existing applications to make use of compiled code in the persistent store.

- There should be little run-time penalty for code that does not deal with persistent objects.
- LIFE interpreter changes should be kept to a minimum.
- The prototype is to be built in a modular fashion so that different fetching and storing alternatives can be explored. This will enable different strategies to be tested to determine which one gives a better performance.

All of these goals entail making persistent data easy to use, easy to extend, and easy to tune. The implementation consequences in meeting the design goals are discussed in the following paragraphs.

Physical I/O: Given that the persistent objects of the program reside on secondary storage, a mechanism is needed to retrieve these objects automatically from the database to achieve transparency. When an object is referenced by the program the system needs to identify whether the object is already in main memory or not, and if not fetch it from the database. To determine this, each reference to non-resident persistent object in virtual memory is associated with a *proxy object* that specifies location of the database object and whether the object has been fetched from the database yet.

Caching: To reduce the performance cost of a persistent system, the object faulted into main memory is cached. This pointer will no longer cause an object fault, although every reference is still subject to a runtime check. This reduces the cost of a database fetch over a period of time, but may clutter the virtual memory over a period of time.

Swizzling: The cached objects can be accessed via a lookup in a resident object-table. To reduce the dereferencing cost of persistent objects, the object faulted into main memory is swizzled (figure 4.5), i.e., the objects' external addresses is mapped to an internal address pointing to the object value. This will avoid a relatively expensive search of the resident object-table each time the object is accessed.

Prefetching: Upon an object reference, the system must also determine how much data to fetch. Fetching only the data needed to execute the current operation will save time and will lessen the cluttering of memory.

4.3.2 System Architecture

Object store

Persistent programming systems are generally supported by an object store [31, 30], a conceptually infinite repository in which objects reside. The LIFE persistent object store provides storage and retrieval of objects, where an object is an uninterpreted byte sequence of virtually unlimited size. The store is designed to be efficient and extensible. Objects are grouped together into the files supported by the operating system.

Access to these objects is via unique *object identifiers* (OIDs). An object's identifier is unique only within the file it is contained in; however an application can have multiple files opened simultaneously, as the *object identifiers* are mapped to globally unique-id values when the objects are referenced in the virtual memory.

The basic unit of data transfer between disk and main memory is a page. LIFE objects are physically grouped together and stored in fixed size pages within a file. Support for sophisticated buffer management is provided. A hash table is provided which takes an object identifier and efficiently determines if the object is resident in main memory. The implementation of the store is similar to the one in [30].

Pages:

A page is the unit of data transferred between disk and main memory.

A page is of fixed size, consisting of page type and the data in it.

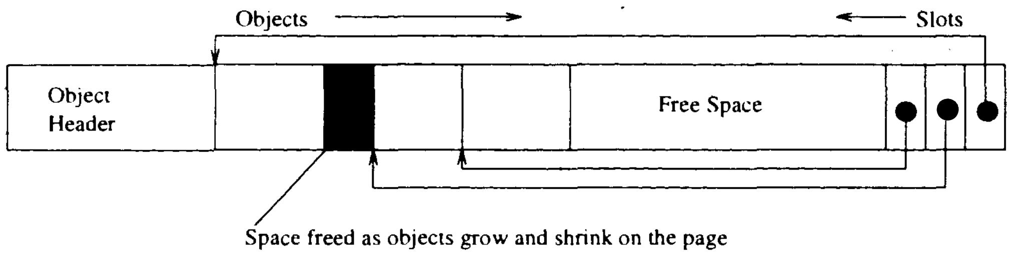

Figure 4.2: Slot page data structure.

The page types are:

File header page: a single page containing meta-information about the file.

Slot pages: a slot page contains objects that can fit into it, header for large objects and meta-information about the page.

Large object page: page containing large object and meta-information for it pertaining to the actual layout of the large object.

Slot page contains the small objects and header for large objects. Slots are used to index and find the actual object. The page is laid out with the slots at the end of page, growing upwards as more objects are added to it. The objects are allocated at the high end of the page following the page header, with the object region growing downwards toward the slot region. The slot page structure is illustrated in Fig. 4.2.

Object:

An object is stored on slot pages and associated with it is a system generated unique *object id* (OID), which allows the object to be located and accessed. The *object id* is an 8-byte quantity consisting of a 4-byte

File ID	**Page number**	**Slot in page**

Figure 4.3: Object-Id structure.

page number, a 2-byte slot within the page and a number to approximate unique ids when the slot space is re-used.

The 2-byte *slot number* provides a pointer to the actual location of the object in the page, allowing an object to be placed anywhere within the page. The corresponding slot is updated when the object is moved around as it grows and shrinks in the page. The *unique-id number* is for re-use of slot space. When a slot is used for the first time the unique-id number is set to one. When an object in the page is deleted this slot is reused and its unique-id number is incremented by one to avoid any dangling references to it. The freed slots are maintained in a linked list, by having the slot-number refer to the next freed slot and the page header containing the first and the last freed slot.

An object's identifier is unique only within the objects file, the *page ID* and the *slot number* together specifying the physical location of the object in the file. However object identifiers are mapped to globally unique identifiers (consisting of *File ID* too) when the objects are accessed. The format of an in-memory OID is shown in Fig. 4.3.

At the store level each object is an uninterpreted container of bytes with an object header attached to it, intended for indicating such properties as the object's length, whether it is small or large object, etc. Fig. 4.4 shows the object format for the storage level.

Internally, the store keeps track of two types of objects- *small objects*, which can fit entirely in a single page, and *large objects*, which are too

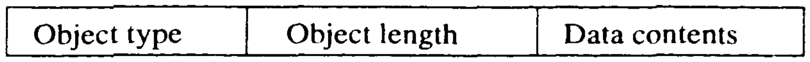

Figure 4.4: Object structure.

large to fit on a single disk page. Small objects are stored in the disk at the location pointed to by the object-id of the small object, while the object-id of a large object refers to a kind of directory called a *large object descriptor*. The contents of a large object descriptor contain meta-information to access the pages holding the object's data.

A page contains a number of objects. An object can grow too big for the page it resides on even though it can still fit into a single page. The object is then moved to another page and a forwarding address pointing to the new location, is left behind in place of the object's original location (object's birth-page). When the object again outgrows its new location, there is no necessity to leave a forwarding address behind at the current physical location. Only the object's birth page contains a reference to its current physical location; all other references point to the object's page. It is this marker that is updated with the new forwarding address reflecting the new location of the object.

It can also happen that a small object grows to the point where it can no longer be contained in a single page. In such a case it is made into a large object, leaving a large-object header on the object's birth-page.

Pointer swizzling in LIFE

The persistent terms in LIFE are stored in an object store. Two address spaces are managed: virtual address space in which objects are directly accessible by the applications and persistent address space of the object store. Objects are transparently moved from one to another on demand.

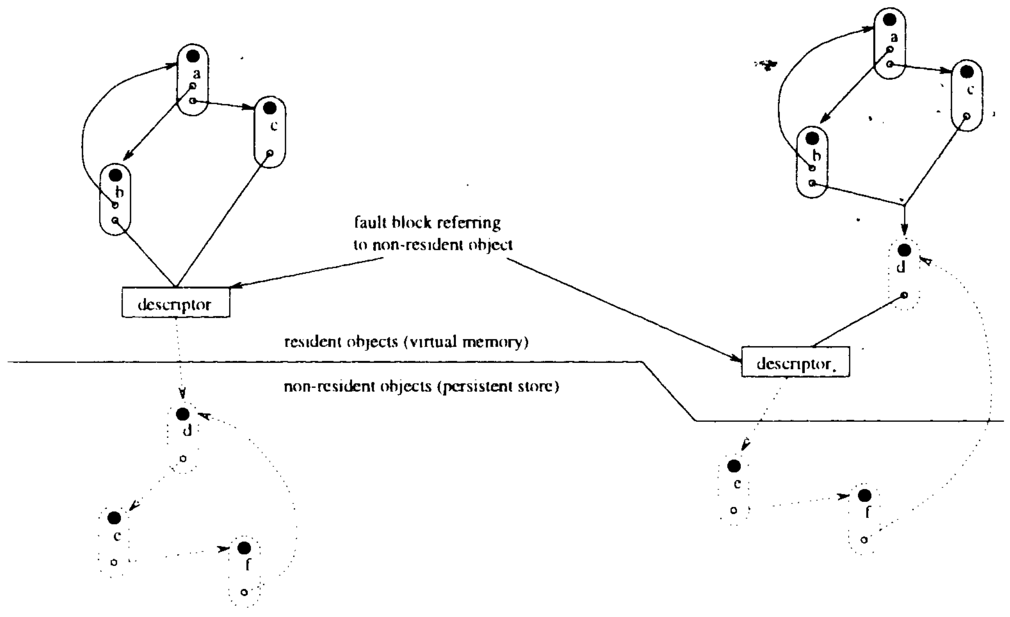
a
b
c
fault block referring to non-resident object
descriptor
resident objects (virtual memory)
non-resident objects (persistent store)
d
c
f
a
b
c
d
descriptor
c
f

Figure 4.5: Pointer swizzling.

Initially a reference to non-resident object consists of a proxy object (object descriptor, figure 4.5). The proxy object contains a pointer to its persistent objects in the secondary storage, and is distinguishable from other objects by its type field.

To speed up access along inter-object references for main memory resident persistent objects, the reference to the proxy object is swizzled into a pointer to the object in main memory. Among the options discussed previously in section 4.2.3, we use lazy swizzling, which swizzles references on demand. This option provides an efficient implementation [42] and is most often used. For portability reason, a software swizzling scheme is used, instead of a hardware scheme.

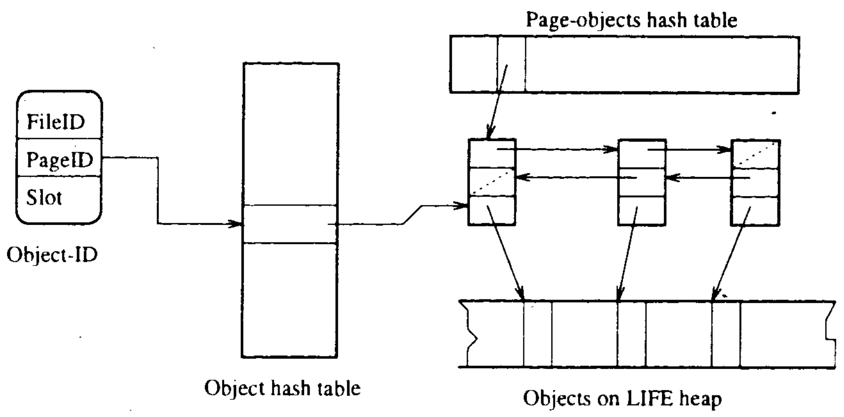

Figure 4.6: Object cache

Buffer management

Performance of persistent applications can be significantly improved, if main memory acts as a cache for disk-based data. Caching data is successful, due to the property of locality which has two aspects:

temporal: The current data item in use will probably be needed by the application again sometime in future.

spatial: When related data items are physically located together (clustering) on disk and brought in as a unit into the main-memory, it is expected that the next data-item is already in main-memory, saving an access to the database.

The buffer manager maintains a page buffer, and an object cache to make use of the property of locality in an application, reducing the swizzling overhead and minimizing disk access.

Page manager:

The unit of data transfer between disk and main memory is a page. The page manager maintains an in-memory chain (*page buffer*) of memory-resident disk pages, indexed by a hash table (*page-table*) on the page-id. Any new page brought in is placed in the middle of the chain. If the page is referenced again it is promoted to the top of the chain. For eviction the pages at the bottom of the chain are chosen.

Whenever a request is made by the object manager to fetch an object from the disk, it looks up the page-table to check first if the page in which the object resides is in-memory. If not on the basis of the OID, the page from the disk is located and loaded into the page buffer pool. A pointer to the object location in the page is then returned to the object manager.

Object manager:

Initially the object manager maintains a chain of *object descriptors*, indexed by a hash table (resident object-table) on the object-id. When an application accesses a non-swizzled persistent reference (OID of the object), the persistent pointer is passed to the object manager. The object manager consults the resident object-table, to see if the object is resident in main memory. If not resident, the persistent object is to be made resident, it gets the reference to the object from the page manager. The object is converted into an internal LIFE format and stored in the LIFE heap, and the mapping from OID to main memory pointer is registered in the object-table. The persistent reference is swizzled to the main memory address of resident objects, to avoid the overhead of consulting the object-table on subsequent access to this reference by the application.

For eviction, the objects at the bottom of the chain are selected. The object descriptors are also chained together on a second hash table according to the disk pages in which the objects lie. This allows updates to objects residing on the same page to be written back to the disk at the same time. Figure 4.6 shows three objects stored on the same disk page.

4.4 Conclusion

We presented an orthogonal persistent LIFE system starting from motivation for the need for it and design principles. A simple lightweight object store was built on top of file systems, as opposed to using a commercial OODBMS. This would meet our design goals of portability, high performance and modularity. The persistent object store caters for object identity, for the storage and retrieval of persistent terms in LIFE.

Besides the orthogonal persistency, another design goal was that performance should compare favorably with non persistent LIFE data. Performance critical issues of detecting database reference, pointer swizzling and caching were addressed. In section 6.1 we have presented a GIS (geographic information systems) application as a natural application for the persistent terms. Performance analysis of the application was done to compare database persistent LIFE with a stand alone LIFE system. The performance of the database persistent LIFE was found to be comparable with stand alone LIFE system and in fact performed better for larger GIS database. The results are shown in section 6.1.2.

A prototype of it has already been implemented and an initial version of it is fully operational except for storage of large objects. The implementation was done partly in C++ language and partly in LIFE and runs on various flavors of UNIX (Solaris, IRIX, Linux, Ultrix and SunOs). The advantages of the database persistency in LIFE can be obtained without any modifications to existing LIFE programs.

Chapter 5

Reverse Compiler

In this chapter, a clustering method is proposed to extract hierarchical categorization of binary relational facts in LIFE. The method gives a polynomial time algorithm for translating a binary relational database into LIFE facts. The algorithm can also be used for:

- concept generation in knowledge systems [57].
- determinization and the minimization of finite-state word and tree automata [58].

This problem was posed by Aït-Kaci [50] and this work is a joint effort of Aït-Kaci [50], Gaur [49] and myself.

5.1 Introduction

Given a binary relational database $R = (A, T)$ where A is the set of attributes and T denotes the tuples over A. If cardinality of A is two, the database can be visualized as a matrix. Let us consider the following example:

$R = (\{a, b\}, \{\{c_1, c_5\}, \{c_2, c_6\}, \{c_3, c_7\}, \{c_1, c_6\}\}$.

R can be represented by a matrix whose rows correspond to the entries in the first column of T and whose columns are the entries in the second column

	c_5	c_6	c_7
c_1	1	1	0
c_2	0	1	0
c_3	0	0	1

Table 5.1: Matrix representation of relation R

of T. If the pair (c_i, c_j) is in T then the corresponding entry in the matrix is 1 else it is 0. The matrix corresponding to R is shown in Table 1.:

Given a 0/1 matrix M, a *rectangle of 1s* is sub-matrix of M composed of all 1s which can be obtained by permuting rows/ columns of M. A *rectangle is maximal* if it is not contained in any other rectangle of 1s obtained by permutation of rows and columns.

As the number of attributes in the relational table is 2, we can represent the database as a matrix. Given a relational matrix M, we are interested in the following questions:

Problem 1: Partition M into maximal rectangles of $1s$ such that the number of rectangles in the partition is minimal.

Problem 2: Given M, find all the maximal rectangles of $1s$ in M.

We are free to permute the rows and the columns of M. Each rectangle in the output to Problem 2 is a concept [57].

5.2 Concept Generation

We will describe an algorithm for solving Problem 2 and show how the merge step in the algorithm can be modified to solve Problem 1. To describe the algorithm we will study a particular class of matrices called *row-convex matrices* and show that there exists a linear time algorithm for both Problems 1 and 2 when the input is restricted to this class. This class forms the base case of our algorithm (figure 5.2). An informal recursive definition of the algorithm is: If the input is *row-convex* (algorithm to generate rectangles for row-covex matrix is shown in figure 5.2) stop, else split the input into two equal sized halves and call the top level routine on the both the parts. The output from both the decompositions is combined using a function called *merge*.

- start point: For each row it refers to the column position of the first 1 in each row.
- end point: For each row it refers to the column position of the last 1 in each row.
- $S = \{S_1, \ldots, S_n\}$ is a sorted list of unique start points of each row.
- For each S_i in S get a list of rows E whose start point is $\leq$ than startpoint of S_i.
    - The rows in E are sorted on their end points from largest to smallest
    - For each row with unique E_i in E, a rectangle is generated consisting of all rows in E whose end point is $\leq$ than end point of E_i.

Figure 5.1: Row Convex Algorithm

A 0/1 matrix M is called *row-convex* if there exists a permutation of the columns of M such that all the ones in every row are consecutive.

Lemma 1 *The number of maximal rectangles in a row-convex matrix* $M \leq n^2$.

Proof: Let the set of n intervals (I) be sorted by their start points, I_0 denote the first interval. We remove I_0 from I, the number of rectangles in $(I_{n-1}$ is denoted by $T(n-1)$. If I_0 is added we add at most n maximal rectangles. Therefore for the recurrence relation is $T(n) = T(n-1) + n$. Hence the number of maximal rectangles $\leq n^2$. ■

Lemma 1 gives us an algorithm for finding all the maximal rectangles in a row-convex matrix M. The algorithm is linear in the number of rectangles outputted. Figure 5.2 shows an informal description of the row convex algorithm coded in *LIFE*.

- Swap columns to make the matrix M row convex as far as possible.
- If the matrix is row convex we are done (Row convex algorithm generates all the rectangles), else
- Rows which have contiguous sequence of 1's is pushed to top of the matrix.
- Split the matrix M into two such that
  - the matrix M1 is a row convex matrix. Row convex algorithm generates all the rectangles for M1.
  - Generate the rectangles for M2.
  - Merge the rectangles generated by M1 and M2.

Figure 5.2: Merge Step

In this section we will describe a divide and conquer algorithmfor solving Problem 2. Figure 5.2 shows an informal description of the algorithm coded in *LIFE*. If the input matrix M is *row-convex* we use Lemma 1 to output all the maximal rectangles. It is easy to determine whether the matrix M is row-convex or not. If M is not row-convex then we partition M into two matrices M_1 and M_2 such that M_1 is row-convex. This is can be achieved by picking the rows which do not have any zeros embedded inside the ones. Next we call the main predicate on M_2 and the output is merged with the maximal rectangles of M_1.

Let R_i denote the set of all maximal rectangles of M_i. Union of M_i possibly contains more maximal rectangles than the union of $R_{i's}$. We now characterize the new maximal rectangles in the union of $M_{i's}$.

Definition 1 Overlap of two Rectangles: *Given maximal rectangles R_1 and R_2* overlap *is defined to be the new maximal rectangle R such that rows of $R = rows(R_1) \cup rows(R_2)$ and columns of $R = columns(R_1) \cap columns(R_2)$.*

Lemma 2 *Union of M_1 and M_2 contains maximal rectangles which can be obtained by* **Overlapping** *$r_l, r_m \mid r_l \in R_i, r_m \in R_j$ and these are the only maximal rectangles which can be added.*

Proof: R_1 and R_2 do not have any rows in common therefore by overlapping them we get a new maximal rectangle C. We will now show that C cannot interact with any $r_i \in R_i, R_j$. Without any loss of generality, assume that r_i belongs to M_1. R can pictorially be represented as shown in Figure 5.3. The only new concepts generated are C_1 and C_2. C_1 can again be divided into upper and lower halfs. The upper half of C_1 is already a maximal concept in R_1 which when combined with some $r_j \in R_2$ will give C_1. C_2 is already in R_1 because it is the overlap of two maximal rectangles T and T_2 in the Figure. We have shown that any new maximal rectangle cannot interact with an old maximal rectangle to generate a new maximal rectangle. From this it follows that no two new maximal rectangles can interact to generate another new maximal rectangle. ■

Next we will give an upper bound on the number of maximal rectangles generated by Lemma 2. We will show that the upper bound is tight.

Lemma 3 *Given a matrix M, the number of maximal rectangles is $\leq 2^n - 1$.*

Proof: Assume that the merge step, partitions M into two matrices of size 1 and $n - 1$. Let $T(n-1)$ denote the number of maximal rectangles in the matrix of size $(n-1)$. Another $T(n-1)$ maximal rectangles can be added in the merge step. Therefore the total number of maximal rectangles is given by

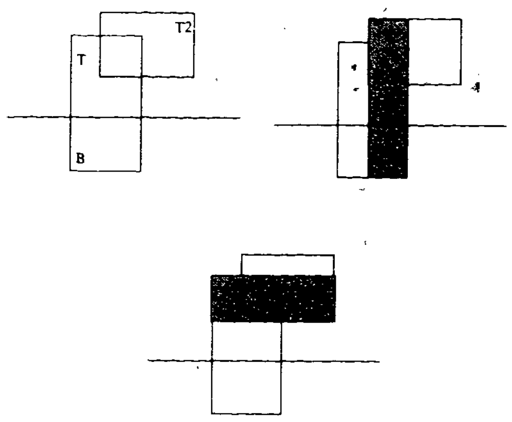

Figure 5.3: Overlap of New and Old Maximal Rectangles

the recurrence relation: $T(n) = 2 * T(n-1) + 1$. Hence the number of maximal rectangles is $\leq 2^n - 1$. ■

Observe that the bound given in Lemma 3 is tight. Let M be a matrix of 1*s* except for the diagonal entries which are 0. For this input the number of maximal rectangles are $2^n - 1$.

In the next section we will show how Lemma 1 and Lemma 2 can be used to cover a matrix by maximal rectangles. This will give us a way of translating a binary relational database into *LIFE* facts (Problem 1). An example of such a translation was provided in section 1.3.3.

5.3 Reverse Compiler

In this section we give a polynomial time algorithm for covering a matrix with maximal rectangles. We now characterize the number of maximal rectangles needed to cover a row-convex matrix. It is easy to observe that covering a bipartite graph with minimum number of complete bipartite subgraphs $K_{m,m}$ can be reduced to covering a matrix M with minimum number of maximal rectangles. Hence the problem of minimizing the size of the covering is $NP-$

Complete. Therefore we will restrict our attention to a greedy covering which is minimal in size.

Now we will show that finding the cover of M with minimum number of rectangles is equivalent to covering a graph with minimum number of maximal cliques. Since the former problem is equivalent to graph coloring it is hard to approximate covering of M [59]. This reduction is stronger than standard NP-Completeness reduction as it tells us about how hard approximating the problem is whereas standard reduction offers no such clue.

We use *CR* to denote minimum cover of a matrix M (with maximal rectangles). *CC* denotes a minimum clique cover of a graph G.

Theorem 1 CR $\iff$ CC.

Proof:

$\Longrightarrow$ Let the vertices of G be (i, j) where i, j are the rows and columns of M. Two vertices $(i, j), (l, m)$ (1s in M) are connected by an edge if $(i = l)$ *or* $(j = m)$ *or* (i, m) & (l, j) are 1's in M. It is easy to see that each maximal rectangle in M corresponds to a clique in G, therefore if we have a minimum cover of M we have a minimum cover of G by cliques.

$\Longleftarrow$ Given a graph G we now construct M. Each vertex i of G is placed on the diagonal (i, i) of M. If (i, j) is an edge in G then we mark (i, j) and (j, i) as 1s in M else the entries are 0. Also, all the diagonal entries of M are 1s. With this construction, if we can find a minimum clique cover G we have a minimum cover of M. ∎

The chromatic number of a graph can be determined by covering its complement with cliques. As Approximate coloring a graph is hard [59], we have approximate covering of a matrix with rectangles is also hard.

Given a row-convex matrix M. Let $S = \cup_r \langle r_{min}, r_{max} \rangle$ | s.t. r_{min} is the first occurrence of a 1 in row r and r_{max} is the last occurrence of a 1 in row r.

Lemma 4 *The number of maximal rectangles required to cover row-convex M is equal to the number of distinct elements in S.*

Lemma 4. gives us the minimum number of maximal rectangles required to cover a row-convex matrix. It takes $O(n^2)$ time to cover a row-convex matrix M with maximal rectangles where n is the size of the row/ column. We can now modify the merge step to produce a cover of M with maximal rectangles. Assume that we recursively found the cover of M_1 and M_2 (the upper and the lower half of M). We now have to check whether each rectangle in the cover of M_1 and M_2 is maximal or not. If it is not maximal then we extend the current rectangle to the maximal rectangle containing it. Thereby generating a cover with maximal rectangles. Observe that the number of maximal rectangles required to cover a matrix M is at most n, where n is the number of rows in the M. Also, this bound is attained for a matrix of all 1's with 0's on the diagonal.

5.4 Conclusion

In this chapter, we addressed the issue of converting binary relational data into LIFE as ψ-terms. We provided a polynomial time algorithm for translating the database into LIFE facts. The algorithm was tested on binary relations and a GUI interface was built to visually verify the correctness of the algorithm.

Chapter 6

Applications

6.1 Geographic Information Systems

Digitization of maps [52, 51] is being currently pursued for effective utilization of the information in various fields: vehicle routing applications, route finding applications, business listings etc. There is lot of focus on digital road maps for vehicle navigation systems. ψ-term's are flexible and useful for representing spatial data in geographic information systems(GIS). The functional and the relational component of LIFE can be used to to express descriptive data and constraints in GIS. It also provides the user with a high level data manipulation language. In this section we consider digital road maps, as an example GIS application in LIFE. The road network is modeled as a persistent ψ-term where edges correspond to road segments and nodes representing road intersections and dead ends. In this way, the spatial relationship between road segments are explicitly retained and can be used for analysis.

In such road networks route finding is a major operation. Displaying of routes for navigation of systems is also needed. A major concern here is degradation of the performance with the increase in network size. In the next section a compression technique is proposed which allows a reduction in the number of nodes and edges in the road network. This reduces the search space size in route finding and display of route. The algorithm finds a near-optimal route from a starting point to the destination point, while improving the efficiency of the route finding algorithm. This problem was communicated to us by G. Misund [48].

In section 6.1.2 we report on the performance of long-term persistency for these terms.

6.1.1 Hierarchical Data Compression

Typically GIS data is in order of Gigabytes, computing shortest path on the actual data can be computationally intensive. In this section we study a compression algorithm for the GIS network which guarantees that the topology of the road network is preserved and routes are approximated reasonably.

The current and anticipated increase in the volume of digital map have revealed two very basic problems in handling the GIS data, namely:

- Visual display of the map: User interfaces are a fundamental component of digital road maps applications, as it would play a critical role in their adoption and success. Effective presentation of route guidance and navigation maps is a non-trivial task due to limited color, small display area and the very nature and size of the data that needs to be dealt with.

- Performance: Digital road maps are the basis for many functions such as positioning, pre-mission route planning, route guidance, map matching etc. Performance of these functions will be a crucial factor with increasing size of the network.

The road network suitably compressed can help improve the visual display of road maps. The map then can be displayed at different levels of details, presenting portions of the map of current interest to the user at a higher resolution. This would improve the user's ability to visually discern the relevant information.

The compression of the network would also help in improving the performance of the queries on the road map such as route finding, which is a frequent operation in such applications.

In this section we provide a hierarchical compression technique to improve the performance of route finding in the road network. The technique is based on a triangle-based edge approximation (observe that other reductions are possible), replacing two adjacent edges (edges having at least one common node) in the graph by a edge (which forms the third side of the triangle) of approximate length. This method for compression of spatial data gives us a hierarchy of triangle-based edge approximations, by applying recursive refinement at each level. For display purpose we have to ensure that the topology of the road network is preserved. For example if there were no cross overs in the original data then the compressed data should also preserve the non-cross over property.

Hierarchical Compression Algorithm

The hierarchical structure is built by selecting a set of adjacent edges in the graph at a given level and applying a recursive compression based on the triangle-based edge approximation.

We number the hierarchical levels from 0 to N, where 0 corresponds to the uncompressed road map and successive refinements are labeled from 1 onwards. Edges and Nodes in the graph will be capital letters. Edges are also given numerical indices, indicating the level in which the edge is present and considered. A road map is described by a graph $G = (V, E_0)$, where V are the vertices of the graph corresponding to road intersections and E_0 are the edges in the graph corresponding to road segments present at level 0 (i.e uncompressed graph).

Following is then a recursive definition of the compression algorithm. To generate the next level i from level $i-1$, consider the edges labeled $i-1$ (i.e present only at level $i-1$). To begin with all the edges of level $i-1$ are added to level i. Now consider any two edges A_i and B_i. If the edges A_i and B_i are adjacent, the two edges are compressed and replaced by a new edge C_i. The two adjacent edges are compressed, only if the newly generated edge C_i does not result in a cross over of the edges in level i and the resulting graph remains connected. The new edge C_i length is the distance between the vertices of the edge, which should be a close approximation to the sum of lengths of the edges compressed.

This scheme will give us a hierarchical structure, with some nodes not belonging to higher levels (this happens when all the edges incident on the node are removed). To determine the shortest route between any two vertices of the hierarchical road network, we find the highest level i in which both the vertices are found. We then compute the shortest path between the two vertices, considering the graph of level i and using the A* algorithm described in the next section. This gives us a near optimal shortest path. More sophisticated schemes for route finding are also possible.

Route Finding

Optimal path planning in a network of road map is one of the basic tasks in the transport industry. For example, to deliver goods from a warehouse to a customer, we need to find the least cost path among the possible set of paths between the two locations. The cost function to be minimized could be any of the parameters such as time, distance etc.

An A* algorithm [53] is used to find the optimal path between any two given points. In this algorithm, a heuristic search function is used to prune the search space, expanding fewer nodes than the popular Dijkstra's shortest path algorithm. The road network consists of nodes and edges. For each node, we associate a cost of reaching the destination from the source. The cost is computed as the sum of the cost to reach a node from the source node (via a particular path) and an estimate of cost to reach the destination from this node. The algorithm to compute the shortest past is as follows:

1. Place the start node on the stack.

2. Pop the first node from the stack.

3. If this node is same as the destination node, we are done and the cost assigned to this node is the smallest cost.

4. Find the neighboring nodes of the node removed from the stack. Estimate the cost for each neighbor node to the destination node. The neighbor nodes are then added to the stack as follows:

   (a) If the neighbor node is found on the stack, the current cost estimate of the node is compared with the previous cost estimate of the node. If the current cost estimate is smaller, update the cost of the node. If the node is not on the stack insert the node in the stack.

   (b) Sort the nodes in the stack in the increasing order of their cost estimate.

5. Repeat steps 2-5.

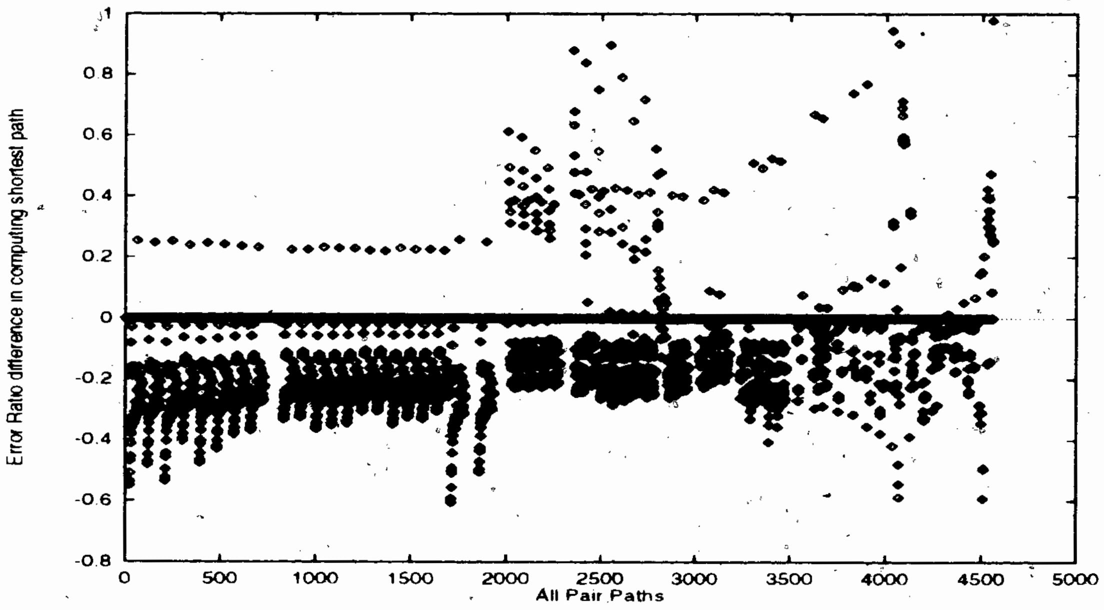

Figure 6.1: Compression obtained on the road map database

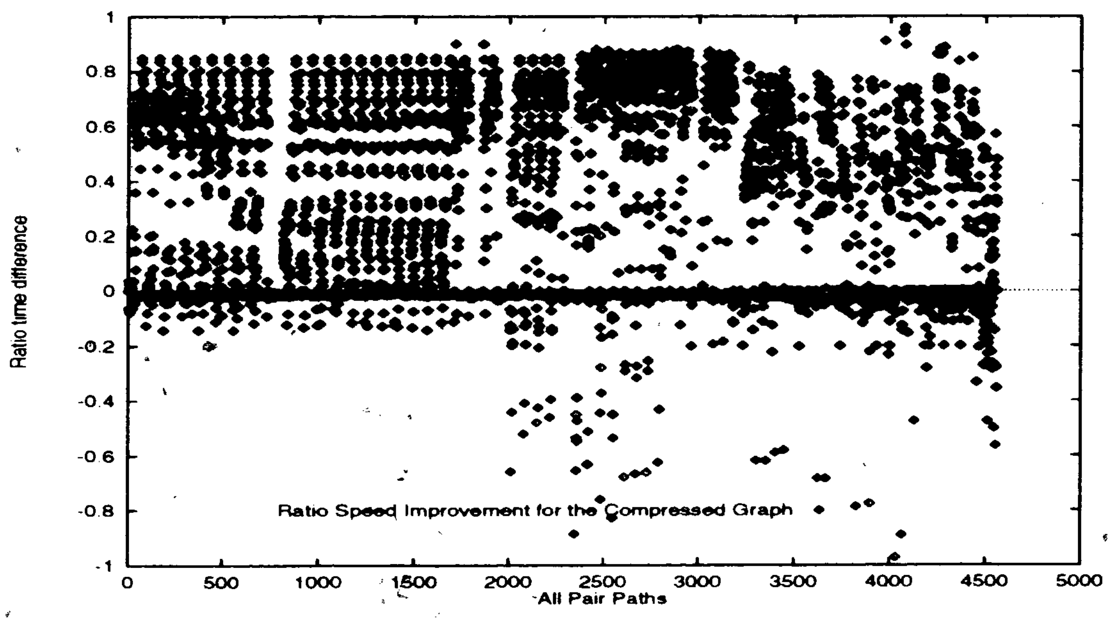

Figure 6.2: Performance gain of the compressed map

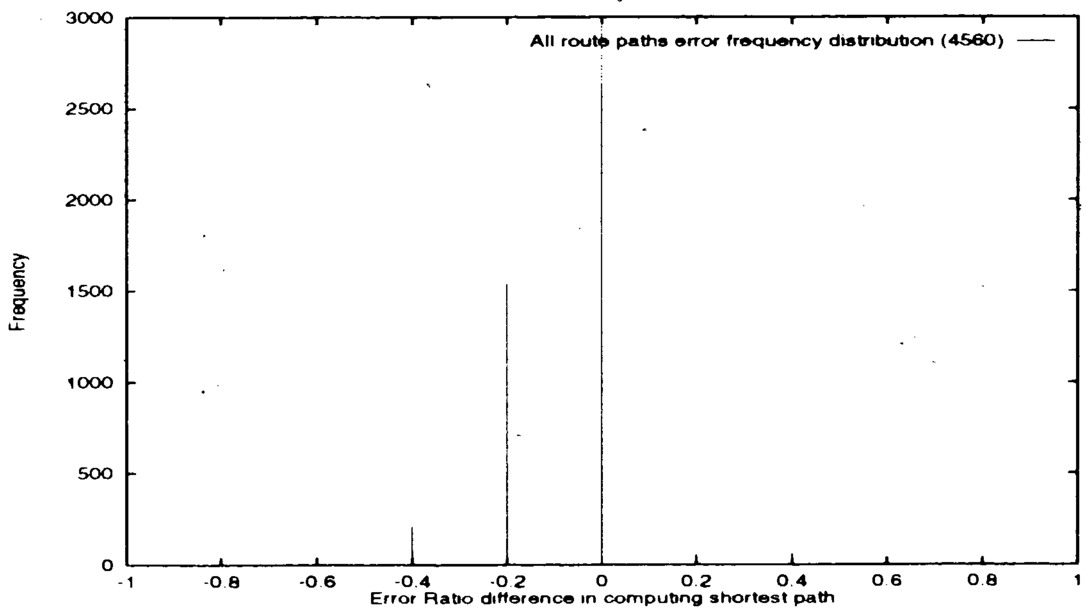

Figure 6.3: Frequency distribution of compression of map

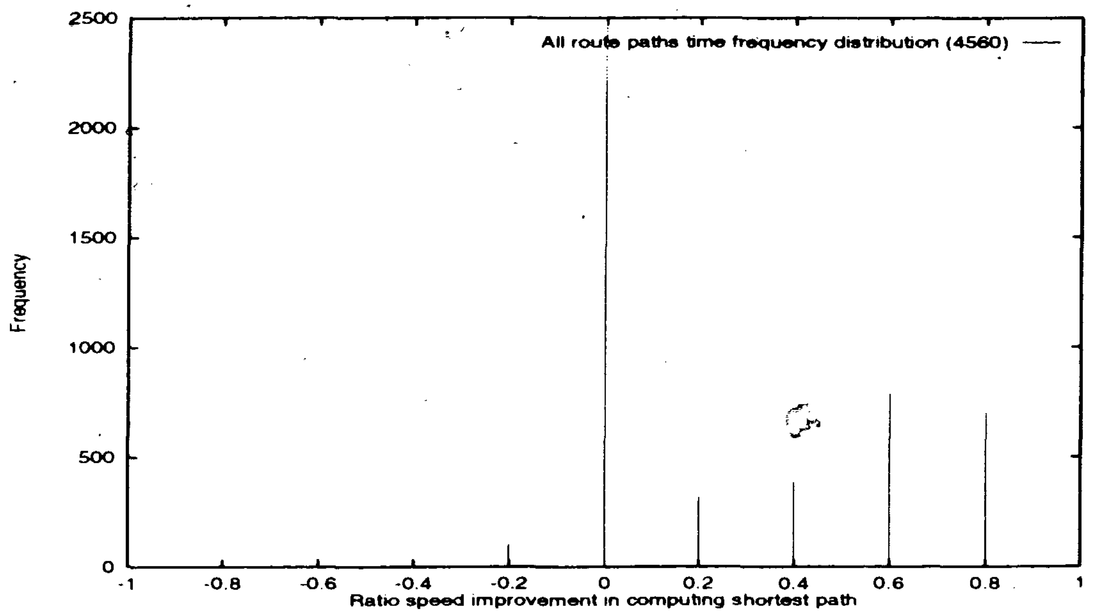

Figure 6.4: Frequency distribution of performance gain of the compressed map

Performance of the Algorithm

In this section we examine the performance of the hierarchical compression algorithm on the real world data of Royken area in Oslo provided by G. Misund [48]. Two factors to be observed are the compression achieved and the variance in shortest paths computed using original and compressed data. We limit ourselves to the smaller data set. This enables us to generate numbers for all pair shortest paths in original data.

The compression algorithm was applied to a road map consisting of 100 edges and 97 nodes. A simple greedy scheme gave a compression of 30% , removing a total of 38 edges. Next we computed the approximate shortest paths between all pairs of nodes in the compressed graph, and compared it with all pair shortest path on uncompressed graph. Figure 6.1 shows the ratio difference between the two paths for all pair of nodes. The percentage speed improvement for the compressed graph is shown in figure 6.2. The frequency distribution of percentage error difference and percentage speed improvement are plotted in figures 6.3 and 6.4. More than 90% of the route paths were computed with an error difference of less than 20%, while 40% of the route paths showed speed improvement of more than 50%.

6.1.2 Performance of the Persistent Database

The performance of long-term persistency for the road map database above has been evaluated and the results are presented in this section. The goal of this study is to analyze the cost associated with persistent data and contrast it with application performance when the entire application database is to be in the virtual memory. For our performance study the system configuration used for the tests is a Dec alpha with 128 Mb of RAM, and 5 Gb of disk space running the OSF 3.0 version of the operating system.

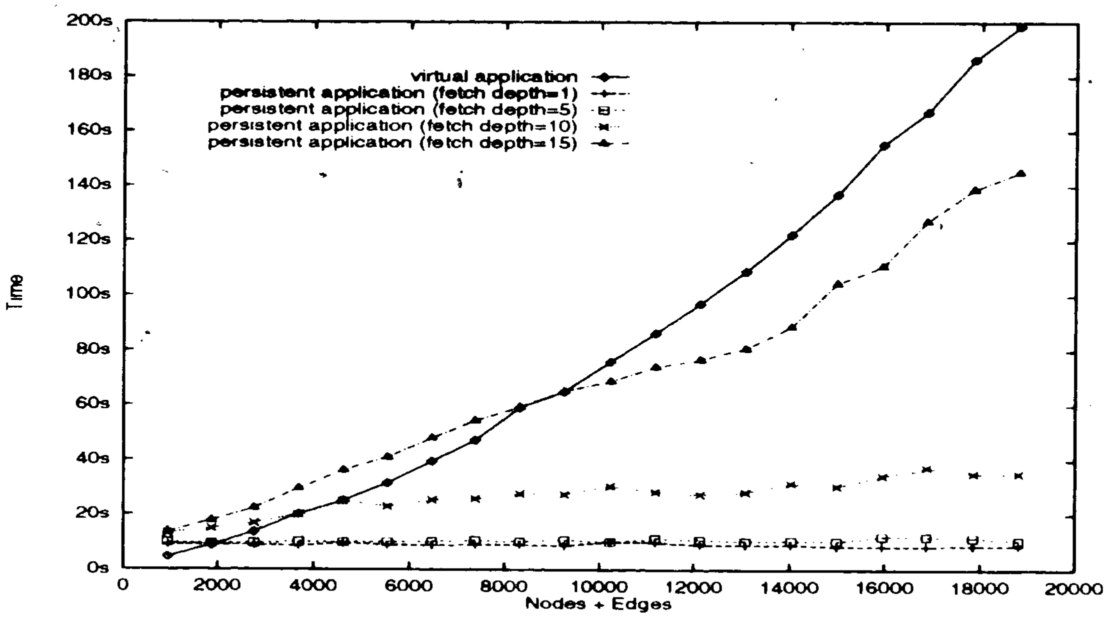

Figure 6.5: Startup times of map

The road map is stored in the persistent database as a persistent ψ-term of nodes and edges. Once the data is stored in the database, the application is run by retrieving only the data currently needed. For the case where the application data can be only in virtual memory, the *virtual* application needs to build the entire road map database, every time it needs to run. When the user starts a query session, it is important that the application start quickly. If the database is persistent, one has to load only the source code needed for querying and not the entire database from the disk. Thus persistent database has the advantage that the road map need not be generated again, every time some computation is to be done on the database. The startup times for the two systems are shown in figure 6.5. The depth of the ψ-term retrieved is varied from 1 to 15 for the persistent application. The system with database persistency starts quickly, regardless of the size of the data when the depth of the ψ-term retrieved is small. For the virtual system, as expected the load time is large (the start time increases rapidly) and worse for the construction of the road map converting raw data to internal memory reperesentation.

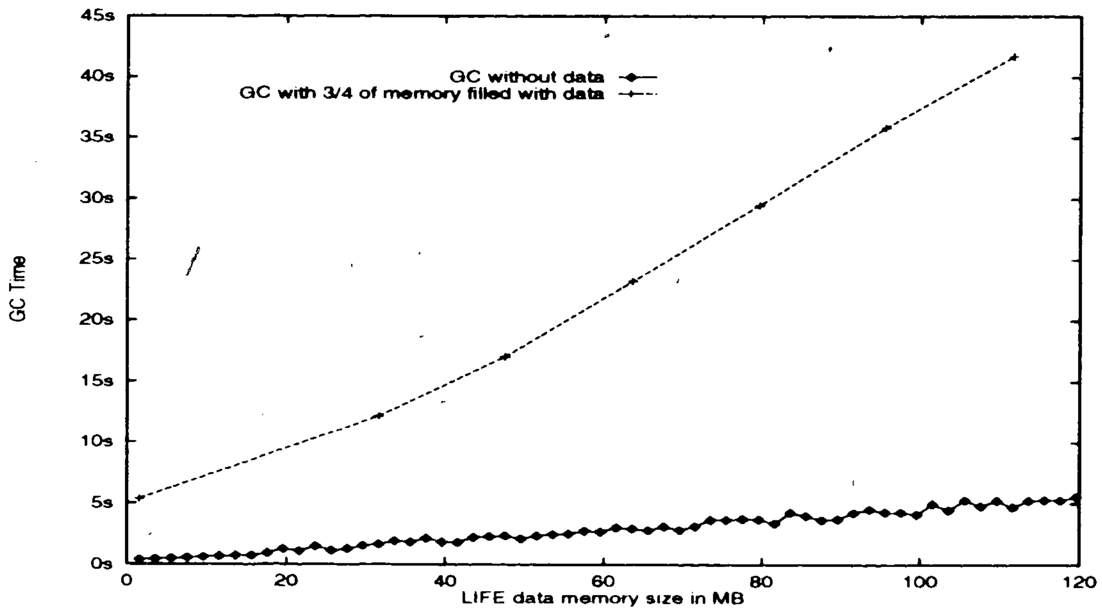

Figure 6.6: Garbage Collection (GC) times in LIFE

LIFE allocates a fixed amount of virtual memory for the application data to be manipulated. Garbage collection (GC) is done when the allocated memory is used up. Garbage collection in LIFE could constitute a significant portion of user time, especially if LIFE is started with large virtual memory. The plots for GC are shown in figure 6.6 for both with very little data in LIFE memory and 3/4 of it filled up with data. In a persistent application, there should be less frequent garbage collection as the application works on a smaller data set, although the database may be large.

Fetching Objects

To test retrieval time, we performed a set of queries on how to get from one point of the road map to another in the shortest possible time using the algorithm described in section 6.1.1. A series of performance tests were run on different size data sets on both the systems. The number of edges of the data set being loaded was varied from 50 to 850.

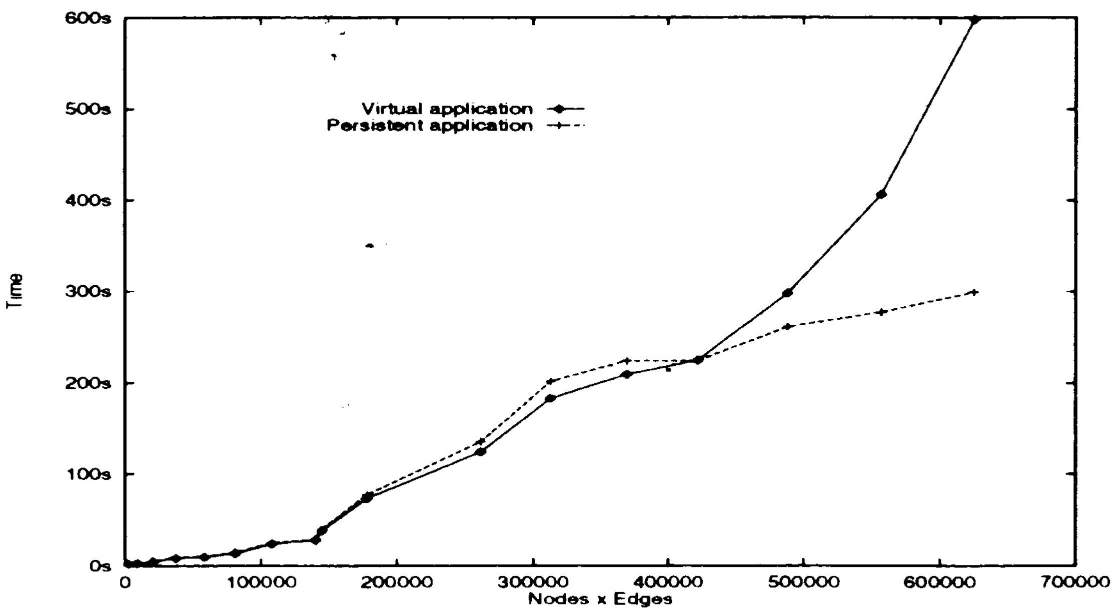

Figure 6.7: Performance of Route Finding Algorithm

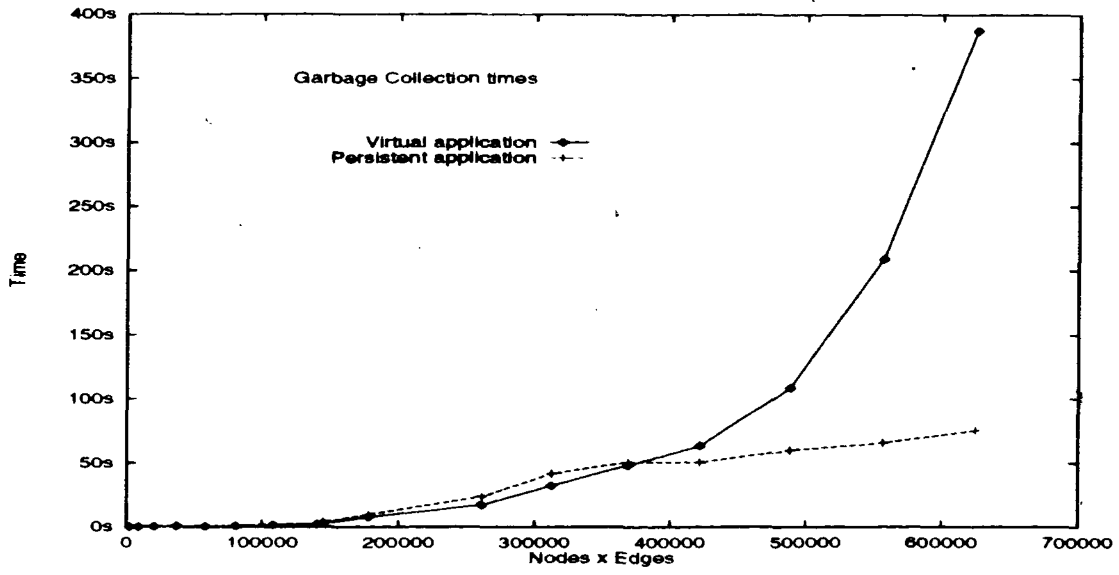

Figure 6.8: GC times in Route Finding Algorithm

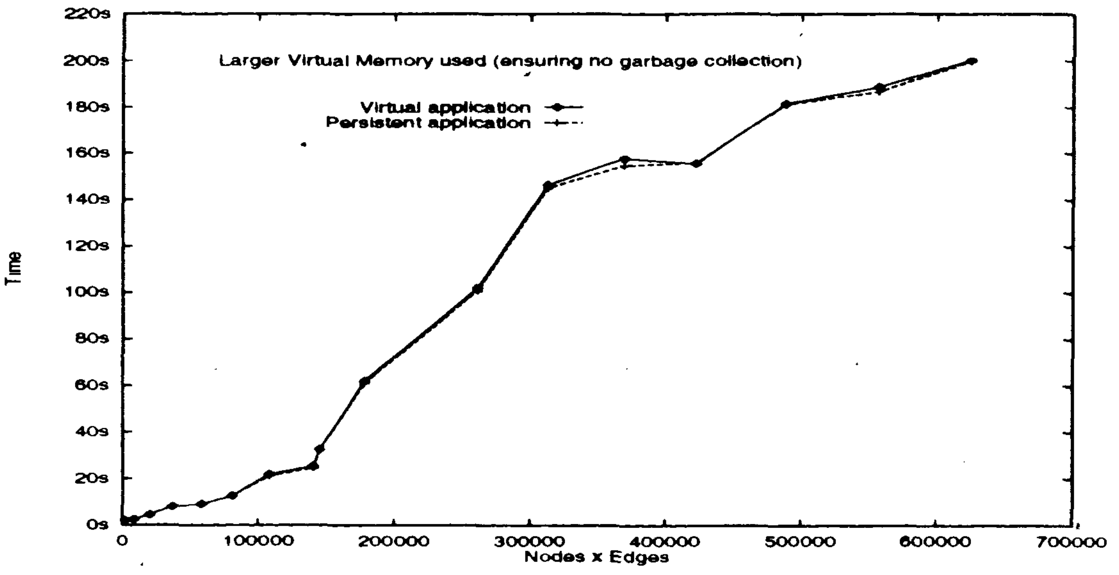

Figure 6.9: Performance of Route Finding Algorithm with larger virtual memory

The plot in figure 6.7 shows the average time taken to execute the set of queries for databases of different sizes ranging from 50 to 1000 edges. All times are an average of 5 readings.

The virtual application performs quiet well, when the data set size is smaller than the LIFE memory size. However as the data set size increases, it starts thrashing due to garbage collection. The time used by the garbage collection is separated out and shown in figure 6.8. Figure 6.9 shows plots of both the system when using a large enough virtual memory to avoid any garbage collections.

6.2 Information Retrieval Systems

In the past several years, there has been a flood of information available over the world wide web. This has resulted in development of many new systems [55, 56] that allow users to search for and access these resources. This rapid growth in the number and size of bibliographic, full text and other electronic information sources, has led to a new problem associated with the search and retrieval these resources: finding information which is of interest to the user. Traditional information retrieval (IR) systems use simple string matching for finding the documents, relying on methods such as statistical measures to define relevance. These systems are quiet effective for some kinds of searching for example known item searching, but when given imprecise information to search for, they may not get truly relevant information. As mentioned earlier, an approach to improve the effectiveness of the information retrieval system is to express the conceptual content of text in a knowledge representation technique [10]. The conceptual representation enables the user to find information, that a user has not named explicitly, but neverthless its relevant to the user.

As an example application, we examine the organizing and searching of bibliographic databases. A knowledge base of bibliographic databases constructed and represented as type hierarchy in LIFE is shown in figure 6.10. The main focus will be on testing the implementation of the LIFE-SQL interface, which has been designed for such large databases.

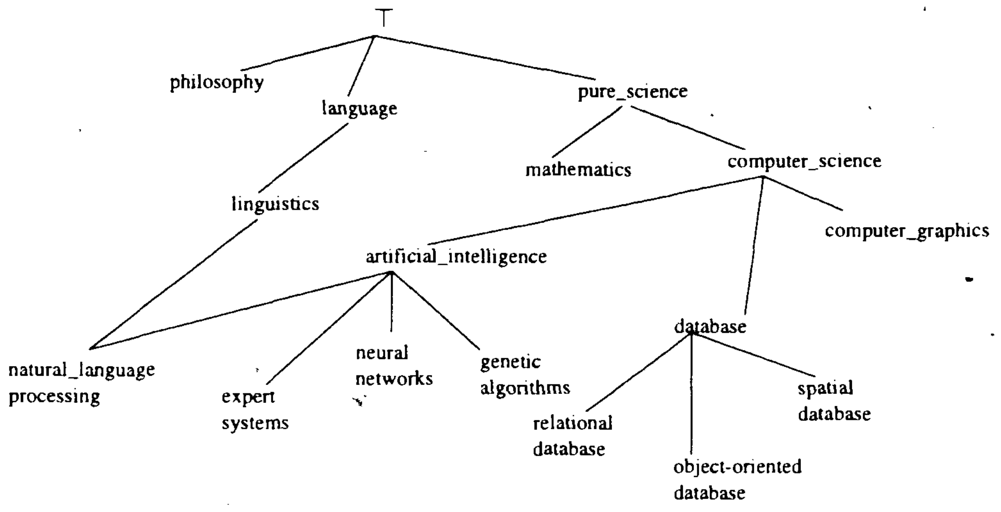

Figure 6.10: Concept hierarchy for bibliographic database

6.2.1 Bibliography Database

For our experiments we got a large collection of bibliographies of scientific literature in computer science from [54]. The collection contains journal articles, conference papers and technical reports in BibTeX format.

A BibTex entry contains information about title, author, keywords, etc. Each entry is represented in LIFE as a ψ-term as:

```
_document ( title ⇒ "Types and Persistence in Database programming",
            author ⇒ (fname ⇒ 'Malcolm', lname ⇒ 'Atkinson'),
            subject ⇒ 'Database',
            contained_in ⇒ journal(acs87),
            keywords ⇒ ['Atkinson', 'Persistence']
          ).

journal ( acs87,
        name ⇒ "ACM Computing Surveys",
        publisher ⇒ 'ACM'
        date ⇒ date(month ⇒ 'June', year ⇒ 1987)
    ).
```

The bibliographic database is compiled into the external database based on the concept hierarchy of symbols in the database.

6.2.2 Knowledge representation using conceptual hierarchy

The knowledge base contains a conceptual hierarchy (figure 6.10, a slightly modified version of figure in [10] on page 258) of the subject matter of the bibliographic entries. The bibliographic entries are frequently searched by their subject matter. The conceptual relationship between subject types of the bibliography database can be represented as a type hierarchy in LIFE. For example the subtype relation **database** $\leq$ **computer_science** encodes the knowledge that **databases** is a subfield of **computer_science**.

The main advantage of such a representation is that we can handle imprecise information in user queries. For example if the user requests documents in linguistics, it will retrieve all documents in natural_language_processing.

6.2.3 Performance Analysis

The results of a series of experiments on the performance of LIFE-RDBMS interface is presented in this section. For all these experiments a tightly coupled architecture is used where the database facts are dynamically retrieved as and when needed by the application. The performance of LIFE-RDBMS has been measured on a Dec alpha with 128 Mb of RAM, and 5 Gb of disk space running the OSF 3.0 version of the operating system. A LRU (least recently used) policy was used in eviction of database facts when LIFE memory was full.

The experiments were divided into three categories, evaluating the performance of:

- Stand alone LIFE with LIFE-RDBMS interface.
- Caching.

Stand alone LIFE with LIFE-RDBMS Interface

One of the main problems in a stand alone LIFE is that the unification mechanism in LIFE forces matching with all the clauses of the fact. If the database is large this could constitute a considerable proportion of processing time, as a result of which the performance would be intolerable. A chief advantage of a tightly coupled LIFE-RDBMS system would be that the working set of facts would be small, resulting in a performance gain.

For the storage of LIFE facts in the database, we had partitioned the facts (see section 3.3) based on the type hierarchy of the symbols occurring in these facts. Each such partition was then stored in a separate relation. This mechanism provides a *concept based clustering* of facts into a single relation. The concept based clustering would improve performance as typically the user is interested in retrieving facts of similar concepts.

We tested the retrieval times of Stand alone LIFE and LIFE-RDBMS interface for single queries. We ran the shallow and deep backtracking queries for both the cases. The shallow and deep backtracking results (in table 6.1) indicate how quickly LIFE unification process can generate answers to a query (LIFE indexes clauses by the functor name of the clause).

Table 6.1: Performance of *LIFE-RDBMS Interface vs Standalone LIFE*

Query	*LIFE-RDBMS*	*Standalone LIFE*	*Cardinality*
query1(shallow backtrack)	0.133333s	0.0333333s	10397
query2(deep backtrack)	0.183333s	1.71667s	10397

Caching

Caching queries and their answers was done to prevent sending the same queries to the database again. A source of such queries is the backtracking mechanism in LIFE. To avoid this we cached both the queries and their answers. Although initially there would be several interactions with the database, caching data could get the working set (which is usually much smaller than the database) running in main memory.

To estimate the speedup that could be achieved we also implemented a system without any caching. We then tested the performance when individual queries are submitted to both the systems. Table 6.2 show the results of cached and non cached systems.

Table 6.2: Performance of *Caching in LIFE-RDBMS interface*

Query	*Caching*	*No Caching*	*Cardinality*
query1	0.25s	0.416667s	10397
query2	0.183333s	0.316667s	10397

6.3 Conclusion

We presented two applications in this chapter to analyze the performance of database interfaces for LIFE. The performance of the persistent store for LIFE was comparable to the stand alone LIFE system. The performance was better for large databases, as there were fewer calls to garbage collection routines. Also the startup times of the applications are small if the database interfaces are used for large applications. The deep backtracking in LIFE resulted in poor performance for large applications. The performance improved when the facts are stored in the database as now the LIFE unification engine had to deal with a smaller set of clauses. Caching the database facts improved the performance by reducing the number of calls to the database.

Chapter 7

Conclusion

7.1 Overview of the System

In this thesis we have presented the details of the implementation of database interfaces for relational and object oriented data in LIFE. The design of the interfaces was motivated by a need for simple and efficient database facilities for LIFE for large applications. For the storage and retrieval of LIFE facts an external RDBMS (SYBASE) was used. As standard SQL-statements are used for the interface, the system is portable and any relational database can be used which provides SQL. For long term persistency of persistent terms an object store was built on top of the operating system's file system. The advantages of these underlying databases can be obtained with no changes to the user programs. A prototype of the two interfaces is working and has been tested on two real world applications.

7.1.1 LIFE-RDBMS System

In Chapter 3 the implementation of a tightly coupled LIFE and a relational DBMS was described. The design enhances the execution speed of LIFE when dealing with large clause sets. We have provided a transparent interface between LIFE and RDBMS. This will allow the user to write whole application in LIFE, without the need to know the RDBMS system. The main problem encountered here was to handle two separate unification environments (before and after retrieval of database facts). We needed to ensure correct backtracking over old and new facts (for example: for efficiency reason LIFE keeps a pointer to the next rule its going to execute).

The interface provides for efficient storage and retrieval of complex objects as flat relations. A concept based clustering of these facts into relations was implemented to improve access time. The database schema generated provided a filtering effect, which then retrieved a smaller resolution set from the database thus reducing a number of unnecessary unifications. The past queries and their answers were also cached in a compact way. This reduced the number of calls to the external RDBMS system and also avoided loading the facts twice. Automatic eviction of cached database facts is also provided when main memory becomes full.

Implementation Improvements

Several aspects of the theory (see section 3.3) upon which this implementation is based upon which were either suboptimal or incomplete were improved upon. We extended the approach to handle multiply inherited types as well. In applications like NLP, coreference constraints in facts occur frequently. We demonstrated how to handle such constraints and store the facts containing such constraints in the relational database. Another aspect of the interface that was improved upon was occurrence of variables in goals.

7.1.2 Persistent Programming in LIFE

We have presented the design and implementation of long term persistency of ψ-terms in chapter 4. It supports the main requirements of orthogonal persistency namely:

- persistency as an abstraction over storage.

- reliable and transparent transfer of persistent terms between long and short term memory.

For storage purposes, a simple lightweight object-store was designed and implemented on top of the unix file system, as opposed to using a commercial OODBMS. This meets our design goals of portability, high performance and modularity. The store caters for storage and retrieval of objects, as an uninterpreted byte sequence. Object-identity is a key concept here in the description of database instances.

We also met our second design goal, namely that the performance of persistent LIFE should be comparable to non-persistent LIFE. The efficiency issues we dealt with here were:

- Identifying database persistent objects
- Pointer swizzling
- Cache management

For detecting database persistent objects, a software scheme was used. Hardware based schemes were avoided mainly for portability reasons. Also previous studies have shown that performance of software based schemes are comparable to hardware based schemes, if not better. Pointer swizzling was employed to minimize the overhead cost of a lookup table when the object is referenced again. This would amortize the cost of swizzling over several references to the same object.

7.2 Reverse Compiler

The problem of converting relational data into LIFE as ψ-terms, was addressed in chapter 5. A clustering method was given to extract hierarchical categorization of relational facts in LIFE. The method gave a polynomial time algorithm for translating a relational database into LIFE facts . The algorithm was tested on binary relations and a GUI interface was built to visually verify the correctness of the algorithm. This translation buys us compact representation of the relational data using the expressive power of ψ-terms (section 1.3.3). Further research needs to be done to extend our approach to n-ary relations.

7.3 Applications

We believe that the combination of LIFE and database systems has definite advantages. As stated earlier, declarative style of programming (mixing functional and relational expression) in LIFE, flexible ψ-term data model and powerful type mechanism of LIFE provide a good platform for applications which require complex data and reasoning power. In order to test the effectiveness of the database interfaces two applications were designed. Our experiments with practical systems show that LIFE is an excellent tool for building real world applications. The GIS application showed that ψ-terms in LIFE provide a flexible data model. The bibliographic database application showed that our system can be used to construct an "intelligent information retrieval systems".

7.4 Performance

We analyzed the database interfaces using these two applications in chapter 6. We measured the performance for various aspects of these interfaces for the two applications, the results of which can be found in section 6.1.2 and 6.2.3. Our main conclusions of these experiments are as follows:

1. LIFE persistent object store offers good performance. The performance of a very data intensive GIS application on the database persistent LIFE was comparable to that on the stand alone LIFE system. The performance for the database persistent LIFE improved for larger databases. It is obvious that the gain was mainly due to the fact that there were fewer calls to GC routines in this case.

2. As expected for both interfaces the start up times of the applications were reduced.

3. For the LIFE-RDBMS interface it was expected that the stand alone LIFE would perform badly for deep backtracking unification. LIFE indexes clauses on the clause functor name, as a result of the linear search it gives poor performance for deep backtracking. Performance improved if the data is stored in an external RDBMS and selectively retrieved.

4. The LIFE-RDBMS interface performance is very good if the retrieved facts are cached. There would be a performance penalty if large number of facts are cached in main memory. This would result from the slow unification of LIFE for deep backtracking queries. This can be reduced by fixing how many database facts are to be in main memory (The interface automatically evicts database facts if the retrieved facts number more than a percentage of total database facts).

7.5 Limitations and Directions for Further Research

The current implementation has a number of limitations and unimplemented features.

- **Performance**: Further detailed experimental study of the database interfaces is required. For instance, the lookup table maintained in LIFE-RDBMS interface for cached facts could be an expensive overhead. A large size lookup table would be generated if the queries to the database return a large number of small sets of tuples. A combination of cached and non-cached facts would give better performance in this case. Comparision to other systems similar to LIFE also needs to be done.

- **Assert and Retract**: The current system cannot handle assert and retract of database facts. The semantics of assert is not well defined, as the user normally does not specify where the facts are to be inserted. The interface associates each qualified segment with a particular file. A solution would be to create qualified segments on the fly. For updates and retracts, a unique *id* needs be associated with every database fact. While retraction is then straight forward, updates will require to find the appropriate qualified segment to move into from its previous qualified segment. This would require dynamically changing the schema.

- **Object Clustering**: Object Clustering is important, so as to co-locate objects that are referenced together and thus attempt to avoid performance penalty in disk *I/O*. This would also improve memory usage (as less database pages need to be buffered). The persistent ψ-terms in LIFE provide for explicit representation of links among objects, allowing navigation through these links for data retrieval. Naturally for better performance reasons its crucial that the clustering algorithms be designed based on the graph structure of the ψ-terms (Breadth First Search, Depth First Search). Various other techniques of clustering based on inheritance and structure semantics, gathering statistical information from workload traces, etc needs to be investigated to find a technique most suitable for persistent ψ-terms.

Other database concepts such as transaction control, data security and recovery, indexing, etc would need further research expecially in the context of persistency in LIFE.

7.5.1 Data Mining

Data mining extracts knowledge from databases, an application LIFE is suited for. An inheritance hierarchy of classes constructed on the basis of the contents of the objects offers a powerful system for representing knowledge. The reverse compiler technique automatically extracts concepts as an inheritance hierarchy from relations by searching for regularities among the unclassified objects. While this technique can be used by itself providing automatic knowledge extraction, it should be possible to combine it with information stored in the relational schema, functional dependencies and other constraints on the database. Existing data mining techniques such as learning from examples could be used along with the reverse compiler technique to provide for a better understanding and design of algorithms to search for knowledge in databases.

7.5.2 Heterogeneous Knowledge Bases

LIFE provides for several knowledge representation techniques. A direction of research that could be considered is to provide a common interface to existing knowledge bases. Knowledge Interchange Format (KIF) is a formal language for interchange of knowledge between disparate programs. A LIFE interface to KIF could then provide for combining heterogeneous knowldge bases.

References

[1] Marcel Holsheimer, Rolf A.de By and H. Aït-Kaci. *A Database Interface for Complex Objects.* Logic Programming - Proceedings of the Eleventh International Conference on Logic Programming, pp. 437–455, 1994.

[2] Marcel Holsheimer. *LIFE-WISDOM, a database interface for the LIFE system.* Master's thesis, Computer Science, University of Twente, Enschede, The Netherlands, 1992.

[3] H. Aït-Kaci and R. Nasr. *LOGIN: a logic programming language with built-in inheritance.* Journal of Logic Programming, 3(3), pp.185–215, 1986.

[4] H. Aït-Kaci. *An algebraic semantics approach to the effective resolution of type equations.* Theoretical Computer Science, 45, pp. 293-351, 1986.

[5] H. Aït-Kaci and A. Podelski. *Towards a meaning of LIFE.* PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, France, 1991.

[6] H. Aït-Kaci, Richard Meyer and Peter Van Roy. *Wild LIFE, Available at URL: http://www.isg.sfu.ca.*

[7] H. Aït-Kaci, R. Nasr. *Le Fun: Logic, equations, and Functions.* Proceedings of the ACM Symposium on Logic Programming, pp. 17-23, San Francisco, September 1987.

[8] H. Aït-Kaci and R. Nasr. *Integrating Logic and Functional Programming.* Lisp and Symbolic Computation 2, pp, 51-89, 1989.

[9] H. Aït-Kaci and A. Podelski. *Functions as passive constraints in LIFE.* PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, France(1992).

[10] H. Aït-Kaci and R. Nasr et al. *Implementing a Knowledge-Based Library Information System with Typed Horn Logic.* Information Processing & Management, 26(2), pp.249–268, 1990.

[11] H. Aït-Kaci, Patrick Lincoln. *LIFE, a Naturla Language for Natural Language.* T.A. Informations, revue internationale du traitement automatique du language, 30(1-2), pp. 37-67, 1989.

[12] Richard O'Keefe. *The Craft of Prolog.* The MIT Press, Cambridge, MA, 1990.

[13] M. Minsky. "*A framework for representing knowledge*". In The Psychology of Computer Vision, P. Winston, editor. McGraw Hill pp.211-277, 1975.

[14] Stefano Ceri, Georg Gottlob, and Gio Wiederhold. *Logic Programming and Databases.* Springer Verlag, Berlin, Germany, 1990.

[15] F. Gozzi, M. Lugli and Stefano Ceri. *An Overview of PRIMO: A Portable Interface between Prolog and Relational Databases.* Information Systems, Vol 15, No 5, pp. 543-553, 1990.

[16] Stefano Ceri, Georg Gottlob, and Gio Wiederhold. *Interfacing Relational Databases and Prolog Efficiently.* Proc. of the 1st International Conference on Expert Database Systems, pp. 141-153, April 1986.

[17] Stefano Ceri, Georg Gottlob, and Gio Wiederhold. *Efficient Database Acess from Prolog.* IEEE Transactions on Software Engineering, pp. 153-164, February 1989.

[18] Matthias Jarke, Jim Clifford, and Yannis Vassiliou. *An Optimizing Front-End to a Relational Query System.* ACM sigmod, pp 296–306, June 1984.

[19] Shalom Tsur. *LDL - A Technology for the Realization of Tightly Coupled Expert Database Systems.* IEEE Expert, 1988.

[20] S. Ghosh, C.C. Lin and T. Sellis. *Implementation of a Prolog-INGRES Interface.* SIGMOD Record, Vol 17, No 2, june 1988.

[21] Malcolm P. Atkinson and O. Peter Buneman. *Types and Persistence in Database programming Languages.* ACM Computing Surveys, Vol. 19, No.2, June 1987.

[22] M. P. Atkinson, K. Chisholm and P. Cockshott *PS-Algol: An Algol with a Persistent Heap.* ACM SIGPLAN Notices, Vol 17(7), July 1982.

[23] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey and R. Morrison. *Persistent Object Management System.* Software Practice and Experience, ACM CR 8408-0627, Vol 14(1), January 1984.

[24] A. Dearle, R. C. H. Connor, A. L. Brown, R. Morrison. *Napier88 - A Database Programming Language?.* 2nd International Workshop on Database Programming Languages, Morgan Kaufmann, Salishan Lodge, pp 179-195, 1989.

[25] A. Dearle, A. L. Brown. *Safe Browsing in a Strongly Typed Persistent Environment.* Computer Journal 31(6), pp 540-544. 1988.

[26] R. Morrison, R. C. H. Connor, Q. I. Cutts, G. N. C. Kirby. *Persistent Possibilities for Software Environments.* The Intersection between Databases and Software Engineering, pp 78-87, IEEE Computer Society Press, 1994.

[27] Joel E. Richardson and Michael J. Carey. *Persistence in E Languages: Issues and Implementation.* Software-Practice and Experience, Vol. 19(12), pp 1115-1150, Dec 1989.

[28] Joel E. Richardson, Michael J. Carey and Daniel T. South. *The Design of the E Programming Language.* acm toplas, pp 494–534, Vol 15(3), July 1993.

[29] M. J. Carey and D. J. DeWitt and S. L. Vandenburg. *A Data Model and Query Language for EXODUS.* ACM sigmod, pp 413–423, June 1988.

[30] M. J. Carey, D. J. DeWitt and Joel E. Richardson. *Storage Management for Objects in EXODUS.* Proceedings of the 12th international conference on very lage databases, 1986.

[31] Eugene J. Shekita, Michael J. Zwilling. *Cricket: A Mapped, Persistent Object Store.* Tech-report 956, Computer Sciences Department, University of Wisconsin-Madison, August 1990.

[32] Paul Adams, Marvin H. Solomon. *An Overview of the CAPITL Software Development Environment.* Tech-report 1143, Computer Science Department, University of Wisconsin-Madison, April 1993.

[33] Paul Adams, Marvin H. Solomon. *POL: Persistent Objects with Logic.* Tech-report 1158, Computer Science Department, University of Wisconsin-Madison, June 1993.

[34] Seth John White. Pointer Swizzling Techniques for Object-oriented Database systems. Phd Thesis, 1994, Universisty of Wisconsin Madison.

[35] R. Agrawal and N. H. Gehani. *Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++.* 2nd Int'l Workshop on Database Programming Languages, june 1989.

[36] R. Agrawal and N. H. Gehani. *ODE (Object Database and Environment): The Language and the Data Model.* ACM SIGMOD RECORD, Vol 18(2), June 1989.

[37] A. Biliris, N. Gehani, and S. Dar. *Making C++ Objects Persistent: Hidden Pointers.* Software Practice and Experience, 1993.

[38] A. Dearle. *On the Construction of Persistent Programming Environments.* PhdThesis, University of St Andrews, 1988.

[39] A. Albano and G. Ghelli and R. Orsini. *The Implementation of Galileo's Persistent Values* Data Types and Persistence, Springer-Verlag, pp 253–263, 1988.

[40] P. O'Brien and B. Bullis and C. Schaffert. *Persistent and Shared Objects in Trellis/Owl.* Proc. Int'l Workshop on Object-Oriented Database Sys, sep 1986.

[41] C.L. Chang and A. Walker. *PROSQL: A Prolog programming interface with SQL/DS.* Proceedings First Int'l Conference on Expert Database Systems, 1986.

[42] Antony L.Hosking and J. E. B. Moss. *Object Fault Handling for Persistent Programming Languages: A Performance Evaluation.* OOPSLA 93, Eighth Annual Conference on Object-oriented Programming systems, Languages, and Applications, Vol28, Oct 1993.

[43] J. E. B. Moss. *Working with Persistent Objects: To swizzle or Not to Swizzle.* IEEE Transactions on Software Engineering, 18(8), pp. 657-673, August 1992.

[44] J. E. B. Moss and Anthony L. Hosking. *Expressing Object Residency Optimization Using Pointer Type Annotations* Persistent Object Systems, pp. 3-15, Tarascon 1994.

[45] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In Proceedings of the Fifth International Workshop on Persistent Object Systems, pp. 11-33, September 1992.

[46] C. Lamb, G. Landis, J.orenstein and D. Weinreb. *The ObjectStore Database System.* Comm. ACM 34, 10, pp. 50-63, October 1991.

[47] Shinji Suzuki, Masaru Kitsuregawa and Mikio Takagi. *An Efficient Pointer Swizzling Method for Navigation Intensive applications.* Persistent Object Systems, pp. 79-95, Tarascon 1994.

[48] Gunnar Misund. *Personal Communication* SINTEF Informatics, e-mail: gmi@si.sintef.co.

[49] Dayaram Gaur. *Personal Communication* Simon Fraser University, e-mail: gaur@cs.sfu.ca.

[50] Hassan Aït-Kaci. *Personal Communication* Simon Fraser University, e-mail: hak@cs.sfu.ca.

[51] Hirofumi Ohnishi, Isao Ogawa and Fuminori Morisue. *Map Database Generation System for In-Vehicle Navigation System* Vehicle Navigation and Information Systems Conference Proceedings, IEEE, pp. 607-612, 1994.

[52] Masao Shibata and Yasuomi Fujita. Current Status and Future Plans for Digital Map Databases in JAPAN Vehicle Navigation and Information Systems Conference Proceedings, Ottawa, IEEE, pp. 29-37, 1993.

[53] T.A. Yang, S.Shekhar, B.Hamidzadeh and P.A. Hancock. *Path Planning and Evaluation in IVHS Databases.* Intl. Conf. on Vehicle Navigation & Information Systems, IEEE, pp. 283-290, October 1991.

[54] AlfChristian Achilles. *Bibliographic Databases.* Available at URL: http:liinwww.ira.uka.de/bibliography/index.html.

[55] GLIMPSE. *A tool to search entire file systems.* Available at URL: http:liinwww.ira.uka.de/bibliography/index.html.

[56] *Yahoo.* Available at URL: http://www.yahoo.com/.

[57] Rudolf Wille. *Concept Lattices and Conceptual Knowledge Systems.* Computer Math. Applic. Vol 23, no. 6-9, pp. 493-515, 1992.

[58] Bruno Courcelle, Damian Niwinski and Andreas Podelski. *A Geometrical View of the Determinization and Minimization of Finite-State Automata.* Mathematical Systems Theory 24, 117-146, 1991.

[59] Sanjeev Arora. *Probabilistic Checking of Proofs and Hardness of Approximation Problems.* CS-TR-476-94, CS Division, UC Berkeley, August 1994.

[60] Stewart M. Clamen. *Data Persistence in Programming Languages A Survey* Tech Report, CMU-CS-91-155, School of Computer Science, Carnegie Mellon Universisty, Pittsburgh, 1991.