Journal of Universal Computer Science, vol. 14, no. 20 (2008), 3333-3357
submitted: 5/7/08, accepted: 24/8/08, appeared: 28/11/08 © J.UCS

UCL-GLORP—An ORM for Common Lisp

António Menezes Leitão
(INESC-ID/Technical University of Lisbon
Rua Alves Redol, n. 9, Lisboa, Portugal
aml@gia.is.utl.pt)

Abstract: UCL-GLORP is a Common Lisp implementation and extension of GLORP (**G**eneric **L**ightweight **O**bject-**R**elational **P**ersistence), an Object-Relational Mapper for the Smalltalk language. UCL-GLORP is now a mature framework that largely extends GLORP and that takes advantage of some of Common Lisp unique features. This paper illustrates UCL-GLORP and discusses some of the challenges that we faced in order to find suitable replacements, in Common Lisp, for some of the more esoteric features of Smalltalk that were explored by GLORP.
Key Words: Object-relational mapping, Common Lisp, Smalltalk
Category: D.1.5, D.2.2, D.3.3, H.2

1 Introduction

A large fraction of modern applications need to store information in some persistent form. Relational databases are the dominant technology for providing data persistence and, in many cases, they are also a requirement.

Being forced to store all data in a relational model doesn't mean that the application can not be programmed in a modern object-oriented style. For all mainstream object-oriented languages there exist one or more Object-Relational Mappers (ORM) that can be programmed to transform data from an object-oriented model into a relational model.

In the Common Lisp camp, however, the situation is not so good. In fact, until very recently, only two of the several database interfaces available for Common Lisp include some ORM capabilities: Common SQL[Lev02] and CLSQL[Ros07]. CLSQL can be seen as a free replacement for Common SQL because it implements the same functional and object-oriented interface to the relational operations. Given the similarity between Common SQL and CLSQL, it is sufficient to describe one of them and we will focus on CLSQL because, contrary to Common SQL, it is freely available and runs in all major Common Lisp implementations.

Unfortunately, CLSQL does not provide many of the important features identified by Fowler[Fow05]:

– It doesn't properly implement the Identity Map pattern so it doesn't preserve the identity of loaded objects implying that an object has as many copies as the number of times it was loaded from the database. CLSQL does implement a cache where objects are related to the queries that loaded them but a different query that happens to return some previously loaded object will not notice it. Besides the time and memory waste, this creates severe identity problems such as inconsistent updates to the "same" object.

- It doesn't implement the Unit of Work pattern, forcing the programmer to either manually save all updated objects or else to rely on CLSQL's automatic save mechanism that occurs on every slot update and that causes performance problems due to the amount of database calls.
- It doesn't implement the concept of Object Transaction, meaning that if a database transaction fails while updating some rows, the mapped objects in the application no longer reflect their last saved state. An Object Transaction provides the same purpose as a database transaction but on the object level, thus maintaining the consistency between them.
- It doesn't implement the Optimistic Offline Locking pattern that is based on the number of modified rows. This last functionality can easily be added to CLSQL (we did it) but it is harder to automatically consider it for the detection of concurrent updates and the necessary signaling of the corresponding object transaction failure.

As a result, CLSQL doesn't qualify as a proper ORM. Given the huge amount of effort that is required for developing an ORM from scratch, we decided to adopt a different strategy based on the translation of some already developed ORM from its original programming language to the Common Lisp language. After looking for a sufficiently developed ORM that was available with an adequate license, we end up selecting GLORP—the **G**eneric **L**ightweight **O**bject-**R**elational **P**ersistence[Kni00].

In the next section we will briefly highlight some of the more important characteristics of GLORP. Then, in section 3, we will discuss UCL-GLORP, our rewrite and extension of GLORP for the Common Lisp language. Section 4 will discuss the problems found and the solutions adopted and, finally, section 5 will present the conclusions.

2 GLORP

GLORP is an open-source object-relational mapping layer for Smalltalk running in several different implementations, including VisualWorks, VisualAge, Dolphin Smalltalk and Squeak.

GLORP features a sophisticated mapping layer that uses a declarative approach to map classes to tables, instance variables to columns and references to foreign keys.

Besides being an ORM, GLORP is also a showcase for the principles and patterns that underlie all ORMs. In the next subsections, we will discuss some of those patterns.

2.1 Models and Mappings

GLORP depends on explicit mappings between objects and their database representations. These mappings operate over object models and database table models.

Each object model describes all the attributes of a specific type of object, in particular, all its relevant slots, their datatypes, their readers and writers, etc. The object model is fundamental because it usually contains much more information than what is generally available in a Smalltalk class definition.

Each table model describes all the attributes of a database table, including column names and types, primary keys, foreign keys, constraints, etc. Although the table model can model a legacy database schema, it is also possible to use it to automatically create the corresponding database schema.

Based on both the object models and the table models, several kinds of mappings can be established but two of them are the most used: object slots containing value objects [Fow05] use *direct mappings*, i.e., they are mapped to the corresponding table columns; object slots containing reference objects are mapped to foreign keys, using *one-to-one*, *one-to-many* and *many-to-many* mappings. All these mappings are crucial to translate object operations to database operations.

2.2 Inheritance

The Smalltalk language supports inheritance, a concept that is absent from relational databases. This means that, besides mapping classes to tables, GLORP must also map class hierarchies into tables. There are three different strategies to implement this mapping [Fow05]: (1) *single table inheritance* (also called *filtered mapping*), where all concrete classes of the inheritance tree are mapped to a single table that has columns for all slots of all classes and also an additional column to discriminate to which class a particular row belongs; (2) *class table inheritance* (also called *vertical mapping*), where each class is mapped to a different table that is referenced by the tables of the subclasses; and (3) *concrete table inheritance* (also known as *horizontal mapping*), where each concrete class is mapped to a different table containing columns for all slots of the concrete class, including the inherited ones.

Currently, GLORP only implements the filtered mapping strategy and the horizontal mapping strategy. In both cases, they are implemented using *type resolvers* that are consulted to translate polymorphic queries into queries that examine all the relevant tables. In the case of the horizontal mapping strategy, the type resolver distinguishes types according to the specific values of the discriminating column.

2.3 Units of Work and Transactions

Instead of forcing the programmer to explicitly write code that, for each updated object, also updates the database, GLORP automatically computes the necessary database updates based on the objects that were loaded, created, modified or deleted during a *unit of work*. This not only simplifies the programmer's work but it is also important to allow reordering of the database updates so that all integrity constraints are satisfied.

Besides Units of Work, GLORP also provides transactions at the object level. This means that, for each object that is modified, a shallow copy is created that contains the previous values of the object slots so that, if necessary, each modified object can be restored to its previous state. This mechanism is important to provide consistency between the database and the application level. Whenever a database transaction aborts, the application program is notified and it can choose to also abort, undoing all object changes that were made during the unit of work.

GLORP contains many other features that are worth discussing but that are beyond the scope of this paper. We refer the reader to [Kni00].

3 UCL-GLORP

Given the flexibility and sophistication of GLORP, it was tempting to use it as the basis for a Common Lisp ORM implementation. The plan was to first semi-automatically translate GLORP from Smalltalk to Common Lisp and then to further develop it so that it could take advantage of the new implementation language. However, implementing and extending GLORP in Common Lisp was far from simple and required us to explore less well-known features of the Common Lisp language. At times, we had the feeling that we were "pushing the envelope" of Common Lisp far beyond its original design. We will postpone the discussion of the problems found until section 4 and we will now describe UCL-GLORP, the Common Lisp implementation of GLORP.

UCL-GLORP is an ORM for Lisp. Like GLORP, UCL-GLORP depends on class models and, given the variety of object systems available in the Lisp world, we designed it to be independent of any specific object system, as long as it is class-based. However, some of the more advanced features do depend on the Common Lisp Object System so, in this paper, we will restrict the discussion to the use of UCL-GLORP as an ORM for CLOS.

In spite of being a complete reimplementation with a large set of differences from GLORP, UCL-GLORP owes a lot to its original inspiration and should inherit a similar license. GLORP was licensed under the LGPL(S)—the GNU Lesser General Public License, with clarifications with respect to Smalltalk library usage—but GLORP's main author, Alan Knight, agrees that UCL-GLORP should be licensed according to the LLGPL, the Lisp Lesser GNU Public License[Inc00].

In the next sections we will give an overview of UCL-GLORP and we will discuss some of its more interesting features.

3.1 Models and Mappings

The first step to provide CLOS classes with relational persistence is to define the class models, the table models and the mappings between them. The most flexible approach is to manually specify that information, allowing complete freedom over table and column names, types, indexes and constraints. In many cases, however, there is a strong correlation between the CLOS classes and the database schema. For these cases, UCL-GLORP is capable of inferring models and mappings strictly from plain CLOS classes, as long as the Common Lisp implementation allows class introspection (e.g., using the CLOS MOP [KdRB91]). We will now demonstrate this capability by modeling in CLOS a small database to keep people names and their home address:

```
(defclass person ()
  ((name :type string :initarg :name :accessor name)
   (address :type address :initarg :address :accessor address)))

(defclass address ()
  ((street :initarg :street :type string :accessor street)
   (city :initarg :city :type string :accessor city)))
```

It is important to stress that the previous classes are plain CLOS classes: `defclass` was not shadowed and there are no metaclasses involved. However, we included with the slots information regarding their types and these type declarations allow UCL-GLORP to infer not only the types to use in the corresponding database columns but also the relationship between `person` and `address`.

The next step consists of selecting the intended database *platform* (e.g., postgres, oracle, mysql, etc), the intended database *accessor* (e.g., clsql, cl-rdbms, etc), and, finally, the necessary database *login* information for accessing the database.[1]

All these steps can be done using the function `make-clos-session` that also creates a *session* for talking with the relational database:

[1] In the following examples, we will use the postgres platform and the clsql accessor.

```
(defparameter *session*
  (make-clos-session
   :classes '(person address)
   :username "foo" :password "bar" :database "baz"))
```

Just like GLORP, UCL-GLORP can use any legacy database model but can also automatically create the tables using the following expression:

```
(recreate-tables *session*)
```

This causes UCL-GLORP to issue the following SQL commands to the database:

```
CREATE TABLE person (oid serial NOT NULL,name text NULL,address int8 NULL,
  CONSTRAINT person_pk PRIMARY KEY (oid),
  CONSTRAINT person_uniq UNIQUE (oid))
CREATE TABLE address (oid serial NOT NULL,street text NULL,city text NULL,
  CONSTRAINT address_pk PRIMARY KEY (oid),
  CONSTRAINT address_uniq UNIQUE (oid))
ALTER TABLE person ADD CONSTRAINT person_add_to_address_oi_ref1
  FOREIGN KEY (address) REFERENCES address (oid)
```

Note that a *primary key* `oid` (*object id*) column was included on both tables and that a *foreign key* `address` was included in the table `person` so that each person row can reference its address row. These decisions were made automatically by UCL-GLORP but could have been overridden by the user.

In the previous example, the primary key is a surrogate key, automatically generated by UCL-GLORP and it only exists in the database model. It is generally a good idea to also have a slot in the object model that corresponds to this primary key. This can be achieved either by inheriting from a pre-defined UCL-GLORP class that provides such slot or by *annotating* the relevant slot: UCL-GLORP identifies as primary key the first slot in the class precedence list that has a reader named `primary-key`. With this annotation, the programmer can also choose to have some natural key used as primary key. Whenever a class contains a slot with a `primary-key` reader, UCL-GLORP will treat the corresponding table column as a primary key and will not create the `oid` column.

3.2 Storing and Retrieving

Using the created session, it is now possible to give persistence to our objects. This is accomplished using a `with-unit-of-work` form that keeps track of all the manipulated objects during its dynamic scope. At the end, the unit of work computes the necessary changes to the database and starts a database transaction to persist those changes. Here is one example:

```
(with-session (*session*)
  (with-unit-of-work ()
    (db-persist
      (make-instance 'person
        :name "John Adams"
        :address (make-instance 'address
                   :street "Park Avenue"
                   :city "New York"))))
```

Note that persisting one object entails persisting all objects reachable from it. The generated SQL is the following:

```
BEGIN TRANSACTION
SELECT nextval('address_oid_seq') FROM pg_attribute LIMIT 1
SELECT nextval('person_oid_seq') FROM pg_attribute LIMIT 1
INSERT INTO address (oid,street,city) VALUES (1,'Park Avenue','New York')
INSERT INTO person (oid,name,address) VALUES (1,'John Adams',1)
COMMIT TRANSACTION
```

As is possible to see from the SQL log, UCL-GLORP assigns `oid`s to the rows using database sequences and then inserts them in their respective tables.

It is now safe to shutdown the Common Lisp process. Upon restart, the persisted objects can be reloaded using the `db-read` function. This function accepts many options (some will be described later) but, for the moment, it is sufficient to say that it is possible to read just `:one` instance of the specified class or `:all` stored instances of that class. Here is one expression that returns the previously stored person:

```
(with-session (*session*)
  (let ((p (db-read :one 'person)))
    (describe p)))
```

3.3 Lazy Loading

The evaluation of the previous expression issues the following SQL statement:

```
SELECT t1.oid, t1.name, t1.address FROM person t1 LIMIT 1
```

and prints:

```
#<PERSON @ #x7352377a> is an instance of #<STANDARD-CLASS PERSON>:
 The following slots have :INSTANCE allocation:
  NAME      "John Adams"
  ADDRESS   <unbound>
```

Note that the `address` slot is unbound. This is intended because UCL-GLORP uses *lazy loading* [Fow05]: referenced objects are loaded only when needed.[2] However, on the first attempt to access the currently unbound slot, UCL-GLORP will "resolve" it using another SQL statement:

[2] This behavior can be customized by the programmer on a slot by slot basis.

```
SELECT t1.oid, t1.street, t1.city FROM address t1
 WHERE (t1.oid = 1) LIMIT 1
```

The result is then used to build the appropriate `address` instance that is stored in the previously unbound slot so that future slot accesses behave as usual. If the previous query didn't return anything, the `address` slot would remain unbound and the session would be modified to avoid repeated attempts for resolving the slot.

Again, we should stress that this mechanism does not require any special care from the programmer. All that is needed is to wrap the code in a `with-session` form.

3.4 Null Values

Most object-relational mappers map database `NULL`s into a special value of the programming language. Several Java ORMs use the Java `null` value and the original GLORP uses the Smalltalk `nil` value. For Common Lisp, it is very tempting to use `nil`s and that is what CLSQL does.

It is our belief that this is not the best solution but, for most programming languages, it is the only reasonable option available. To understand the problem, it is relevant to know that, in database terms, a `NULL` value is not a value at all: it represents a missing or unknown value. Unfortunately, translating a database `NULL` into a Java `null` allows a program to manipulate an "unknown value," passing it as argument to method calls, storing it in data structures, and returning it from method calls. It is only later, when the value is used as receiver of a method call, that the "unknown value" causes a null pointer exception. Unfortunately, it is then much more difficult to understand where did the null value come from.

In Common Lisp, using `nil` for representing unknown values is even more problematic than in Java. One obvious problem is caused by the fact that the `nil` value also represents the false logical value and this means that it makes it impossible to distinguish a false value from an unknown value. Another serious problem is that `nil` is (the only) value of type `null`. As was explained in section 3.1, UCL-GLORP uses slot type declarations to infer relations and column types but, from the point of view of the Common Lisp type checker, a slot with declared type `address` can only contain objects of type `address` and it is an error to assign `nil` to such slot.

Fortunately, there is a much better solution that does not depend on using the `nil` object: in Common Lisp, when a slot has a value, the slot is *bound* and reading it returns its value; when a slot does not have a value, the slot is *unbound* and reading it triggers a call to a generic function whose default method signals a condition. UCL-GLORP explores this behavior to map database `NULL`s to unbound slots. This means that the type declarations can be correctly used and it also has the advantage that any attempt to manipulate "unknown values" will be immediately detected.

In some cases, however, we might prefer to represent a database `NULL` using a bound slot with a `nil` value. A proper type declaration for this situation requires a compound type specifier. For example, we could write the following class definition:

```
(defclass person ()
  ((name :type string :initarg :name :accessor name)
   (address :type (or address null) :initarg :address :accessor address)))
```

to represent people that have a known address or that have a `nil` value in place of an unknown address. To deal with this approach, UCL-GLORP also understands `(or` ***`type`*** `null)` type specifiers and, for those slots that have such type declaration, it uses the `NULL`–`nil` mapping technique.

3.5 Relations

In the previous example, each person references just one address but we are not restricted to one-to-one relations. We can also have one-to-many and many-to-many relations. For example, let's suppose each person also has a vector of email addresses. This can be written using a `(vector email-address)` compound type specifier, as follows:

```
(defclass person ()
  ((name :type string :initarg :name :accessor name)
   (address :type address :initarg :address :accessor address)
   (email-addresses :type (vector email-address)
                    :initarg :email-addresses :accessor email-addresses)))

(defclass email-address ()
  ((username :initarg :username :type string :accessor username)
   (host :initarg :host :type string :accessor host)))
```

UCL-GLORP will use the `:type` option in the `email-addresses` slot to infer a one-to-many relation from `person` to `email-address`.[3] This implies that UCL-GLORP will include a foreign key column in the table for email addresses that will point to the person that owns the email address, as is possible to see in the generated SQL for the table `email_address`:

```
CREATE TABLE email_address (oid serial NOT NULL,username text NULL,
  host text NULL,person_email_addresses int8 NULL,
  CONSTRAINT email_address_pk PRIMARY KEY (oid),
  CONSTRAINT email_address_uniq UNIQUE (oid))
ALTER TABLE email_address ADD CONSTRAINT email_addr_to_person_oid_ref1
  FOREIGN KEY (person_email_addresses) REFERENCES person (oid)
```

[3] Besides vectors, UCL-GLORP also recognizes type specifiers for lists of elements, including the more relationally-oriented `(one-to-many` ***`element-type`***`)` and `(many-to-many` ***`element-type`***`)`.

3.6 Updating

After the previous change, we can ask UCL-GLORP to update the database schema, causing it to create a table to contain the email addresses. Now, let's suppose that we want to assign two different email addresses to John and we will also take the opportunity to change the street of the address of John:

```
(with-session (*session*)
  (with-unit-of-work ()
    (let ((john (db-read :one 'person)))
      (setf (street (address john)) "33rd Street")
      (setf (email-addresses john)
            (vector
             (make-instance 'email-address
               :username "012345" :host "freemail.com")
             (make-instance 'email-address
               :username "john" :host "foo.bar"))))))
```

This is where a UCL-GLORP's unit of work becomes very useful: instead of forcing us to manually identify the new and changed objects, it automatically computes all changes and writes the proper sequence of updates and inserts to the database. For the previous example, the generated sequence of SQL statements is the following:

```
BEGIN TRANSACTION
SELECT t1.oid, t1.username, t1.host FROM email_address t1
 WHERE (t1.person_email_addresses = 1)
SELECT nextval('email_address_oid_seq') FROM pg_attribute LIMIT 2
UPDATE address SET street = '33rd Street' WHERE oid = 1
INSERT INTO email_address (oid,username,host,person_email_addresses)
 VALUES (1,'012345','freemail.com',1)
INSERT INTO email_address (oid,username,host,person_email_addresses)
 VALUES (2,'john','foo.bar',1)
COMMIT TRANSACTION
```

Note, in the previous SQL code, that a SELECT statement was issued so that UCL-GLORP could compute the changes to the former email addresses of John.

3.7 Querying

One of the best features of UCL-GLORP is the support for combining "normal" Common Lisp code with database queries. As an example, let's suppose we define a predicate that tests that a given person has an email address on a given host:

```
(defun has-email-on-host-p (person host)
  (some (lambda (address)
          (string= (host address) host))
        (email-addresses person)))
```

Using this predicate, we can collect all people that have email on, e.g., `freemail.com`:

```
(remove-if-not (lambda (person)
                 (has-email-on-host-p person "freemail.com"))
               (db-read :all 'person))
```

The previous code reads `:all` people from the database and then filters those that do not satisfy the predicate. To achieve this goal, the `db-read` call starts by generating a generic SQL query that returns all rows from the `person` table and creates the corresponding objects. Then, for *each* person (with *oid* primary key), the `remove-if-not` function calls the predicate that checks the email addresses, causing a series of SQL queries of the form:

```
SELECT t1.oid, t1.username, t1.host
 FROM email_address t1
 WHERE (t1.person_email_addresses = oid1)
SELECT t1.oid, t1.username, t1.host
 FROM email_address t1
 WHERE (t1.person_email_addresses = oid2)
...
SELECT t1.oid, t1.username, t1.host
 FROM email_address t1
 WHERE (t1.person_email_addresses = oidn)
```

Clearly, this is a waste of resources because the database might contain hundreds of thousands of rows in the `people` table that will have to be loaded and, for each of them, another query will be issued to compute the corresponding rows from the `email_address` table, thus creating a huge amount of objects just to filter them. Besides the space waste, the process will generate a huge amount of traffic between the application and the database, severely impacting the performance.

Fortunately, a simple rewrite of the expression is sufficient to dramatically speed up the process. To this end, the `db-read` function has a `:where` keyword parameter that accepts the exact same function that the `remove-if-not` accepted. Using this `:where` parameter, the previous expression can be rewritten as:

```
(db-read :all 'person
         :where (lambda (person)
                  (has-email-on-host-p person "freemail.com")))
```

The results are exactly the same but they are computed differently. Now, the `db-read` call uses the predicate, not to filter the results, but to compute a single SQL query that returns the relevant people in just one database call:

```
SELECT t1.oid, t1.name, t1.address
 FROM person t1
 WHERE EXISTS (SELECT t2.oid
   FROM email_address t2
   WHERE ((t2.host = 'freemail.com') AND
          (t1.oid = t2.person_email_addresses)))
```

The previous translation uses a subquery but it is also possible to generate an equivalent query that uses a join that, for certain database backends, might have better support. To this end, the programmer can change a flag in the configuration of the database connection and the same `db-read` expression will be translated into the following SQL query:

```
SELECT DISTINCT t1.oid, t1.name, t1.address
 FROM (person t1 INNER JOIN email_address t2
        ON (t1.oid = t2.person_email_addresses))
 WHERE (t2.host = 'freemail.com')
```

Given the fact that database communication is considerably slow and that modern database engines have good query optimizers, any of the two previous SQL queries will likely run much faster, even taking into account the time needed to analyze the predicate and to translate it into an SQL clause. Not every Common Lisp predicate can be translated into SQL but a representative subset can and Common Lisp programmers will like to know that this subset includes closures. For example, let's suppose that `john` references the "John Adams" that lives in the "33rd Street." Then, the following expression returns all people that live on the same street as `john`:

```
(let ((john ...))
  (db-read :all 'person
           :where (lambda (person)
                    (string= (street (address person))
                             (street (address john))))))
```

Note, in the `:where` argument of the `db-read` call, that the function uses the free variable `john`. In this case, the evaluation of the previous expression will make a single database call using the following SQL query:

```
SELECT t1.oid, t1.name, t1.address
 FROM (person t1 INNER JOIN address t2 ON (t1.address = t2.oid))
 WHERE (t2.street = '33rd Street')
```

Again, this query will run much faster than loading all people and then filter them on the application side.

We will discuss the predicate translation process in section 4.

3.8 Sorting

As we explained in the previous section, a query can be executed much more efficiently when the filtering criteria can be translated from Common Lisp to

SQL. Besides reducing the traffic between the database and the application, it allows the database backend to produce better query plans.

Filtering, however, is just one of the operations that can benefit from these optimizations. Sorting is another one.

As an example, let us suppose that, again, we are interested in the people that have email on `freemail.com` but, this time, we also want to sort the result by the city of the home address. This can be done using the Common Lisp `sort` function:

```
(sort (db-read :all 'person
               :where (lambda (person)
                        (has-email-on-host-p person "freemail.com")))
      #'string<
      :key (lambda (p) (city (address p))))
```

Unfortunately, in spite of the improved performance of the `db-read` query, the sorting operation will have very poor efficiency because it will be done on the application side: *for each person* retrieved by the first query, the comparison predicate needs to retrieve, from the database, his/her address, thus emitting a potentially large number of requests to the database.

To solve this problem, UCL-GLORP `db-read` function can also translate sorting operations. These are specified using the `:order-by` keyword parameter:

```
(db-read :all 'person
         :where (lambda (person)
                  (has-email-on-host-p person "freemail.com"))
         :order-by (lambda (p) (city (address p))))
```

Using the previous query, a single SQL statement is generated:

```
SELECT t1.oid, t1.name, t1.address
 FROM (person t1 INNER JOIN address t3 ON (t1.address = t3.oid))
 WHERE EXISTS (SELECT t2.oid
   FROM email_address t2
   WHERE ((t2.host = 'freemail.com') AND
          (t1.oid = t2.person_email_addresses)))
 ORDER BY t3.city
```

Note that an additional inner join was generated in order to allow the city of the address to be used as sorting criteria.

3.9 Eager Loading

We mentioned that UCL-GLORP implements lazy loading as a performance optimization that allows it to avoid loading the objects that are related to a retrieved object.

Sometimes, however, this optimization becomes a *pessimization.* As an example, let's consider again the query that returns all people that has an email address on `freemail.com` but, now, let's also compute the cities where they live:

```
(mapcar (lambda (p) (city (address p)))
        (db-read :all 'person
                 :where (lambda (person)
                          (has-email-on-host-p person "freemail.com"))))
```

Due to the lazy loading mechanism, the `db-read` function returns a set of people with unbound `address` slots. However, immediately after retrieving this set, the application computes the city of the address of each person. This means that, *for each person* in the previous result set, another query will be send to the database to retrieve his address, severely impacting the performance.

To solve this problem, UCL-GLORP also implements the opposite of lazy loading: *eager loading.* Eager loading allows the application to selectively request from the database the necessary information to immediately build related objects. To this end, the `db-read` function has an `:also-fetch` keyword argument that accepts a function that expresses the intended relation.

In order for the current example to take advantage of this feature, we can simply include the relation in the query, as follows:

```
(mapcar (lambda (p) (city (address p)))
        (db-read :all 'person
                 :where (lambda (person)
                          (has-email-on-host-p person "freemail.com"))
                 :also-fetch #'address))
```

The generated SQL is:

```
SELECT t1.oid, t1.name, t1.address, t3.oid, t3.street, t3.city
 FROM (person t1 LEFT OUTER JOIN address t3 ON (t1.address = t3.oid))
 WHERE  EXISTS (SELECT t2.oid
   FROM email_address t2
   WHERE ((t2.host = 'freemail.com') AND
          (t1.oid = t2.person_email_addresses)))
```

In this case, note that the rows from the person table are combined with the corresponding rows from the address table using a left outer join, so that the query does not exclude those who don't have an address.

Lazy loading and eager loading should be used in combination. Lazy loading avoids loading an object that is not necessary for the current computation but it entails a performance penalty if the object becomes necessary later on. Eager loading allows us to selectively annotate the code so that this situation can be avoided. With eager loading, all the objects required for a given computation can be loaded in as few queries as possible.

As with any optimization, eager loading should be applied only after profiling the application. As an example, after profiling a web application that uses UCL-GLORP we identified a fragment of code that, on average, was taking 5.7 seconds to run and allocated more than 70 MB of memory. The fragment in question computed the addresses of all the users that satisfied some arbitrary criteria but the address of each user was lazy loaded. After including the address relation as an `:also-fetch` argument, the same code fragment runs, on average, in 1.8 seconds and allocates only 20 MB of memory.

3.10 Database Evolution

Common Lisp was designed to allow an interactive and incremental development process. For example, `defun` can be used to define a new function or to redefine an already-defined function and `defclass` can be used to define new classes or to redefine old classes. In this last case, an instance update protocol is automatically run so that old class instances get a chance to adapt to the new class definition.

Given the fact that UCL-GLORP can derive a database schema from a set of CLOS classes, it would be highly annoying if that schema had to be entirely regenerated on every class redefinition, forcing the programmer to `DROP` the database and to rebuild it from scratch.

In order to be faithful to the incremental development process allowed in Common Lisp, besides mapping a set of classes into a set of database tables, UCL-GLORP is also capable of mapping a set of class *changes* into a set of database *changes*. To this end, UCL-GLORP regenerates a local copy of the entire database schema after a set of class changes and compares it with the previous database schema, computing the differences. From these differences, a set of SQL statements is generated that `CREATE`s only the new tables, `DROP`s only the obsolete ones and `ALTER`s only those that changed, thus moving the database from its previous schema to the new one.

The programmer should be aware that, in many cases, there might be more than one way to do this. For example, let's consider a class redefinition where a slot was added and another one was removed and let's assume that both these slots are mapped into database columns of the same table. In what regards the database, this change can be accomplished either by dropping the old column and adding the new one, or by adapting the old column so that it becomes the new one. There is a huge difference between these two approaches because the first one might require migration code (or loss of data) while the second one might be entirely managed by the database.

UCL-GLORP is agnostic regarding the correct update path so whenever there are mismatches between the old and new database schema, a condition is signaled and restarts for all possible update paths are established. It is the programmer's responsibility to (1) interactively choose the sequence of restarts that provide the best database update path and (2) provide any necessary migration code.

4 From GLORP to UCL-GLORP

During the reincarnation of GLORP as UCL-GLORP, several problems had to be solved in order to overcome the following differences between Smalltalk and Common Lisp:

– In Smalltalk, methods belong to classes and are dispatched according to the class of the receiver. In Common Lisp, methods belong to generic functions and are dispatched according to the type of all the arguments. This is a huge obstacle for the translation because generic functions require congruent methods, while in Smalltalk methods are independent from each other. In practice, each Smalltalk class provides a namespace for its own methods.

– In Smalltalk, the methods of a class have direct access to the instance variables of the receiver. In Common Lisp, this is not possible but can be emulated using the `with-slots` macro. However, a naïve translation from Smalltalk to Common Lisp might end up inserting a `with-slots` form in every method. Replacing `with-slots` with accessors is also not practical because of potential name clashes between the newly created readers and already existent generic functions.

– Smalltalk method invocation protocol makes it easy to explore the proxy design pattern [GHJV00]. A proxy class can redefine the default behavior for the `#doesNotUnderstand:` message so that every message sent to the proxy can have a response even when not directly implemented in the proxy class. This is used, for example, to implement the lazy loading of an object: the proxy stands for some not yet loaded object until it receives a message that it doesn't understand, causing it to load the object and forward the message. Common Lisp's generic function invocation protocol makes it much more difficult to implement the same design pattern.

– Smalltalk provides distinct true and false values. On the contrary, Common Lisp amalgamates the false value, the empty list and the symbol `nil` and treats all other values as true.

– Smalltalk provides a distinct null value that is used to initialize instance variables. Common Lisp relies on a different mechanism where instance variables either are unbound or are bound to a value and there is a protocol for accessing those variables (called "slots" in Common Lisp parlance).

– In Smalltalk, collections have identity. Adding or removing elements from collections preserve that identity. Although some Common Lisp collections also preserve identity across modifications, the most used collection data type—the list—was not designed to preserve identity. Usually, this is not a problem to Common Lisp programmers because they tend to respect the Law of Demeter [Lie89], meaning that they don't directly manipulate containers stored inside some object. However this law is not consistently enforced in Smalltalk programs, where it is not uncommon to see a collection being passed to a method that then modifies it.

There are many more differences between the two languages but these were the ones that had the biggest impact on the translation of GLORP from Smalltalk to Common Lisp. We will now discuss some of the differences.

4.1 Convention over Configuration

One of the biggest differences between GLORP and UCL-GLORP is in the amount of effort that is needed to map an object model to a relational model. GLORP is implemented in Smalltalk, a language that does not provide any standardized way of including slot type information in a class definition. That is the main reason why the programmer has to explicitly describe such type information in a model of the mapped class that must be synchronized with the class itself. The same problem happens with the table model and with the mappings that must be established between the class model and the table model: they must be hand-written and they must be consistent between each other.

Writing all these models and mappings quickly becomes an annoying task for the programmer and a potential source of bugs. They are also difficult to maintain. A single inconsistency between any of them can cause hard-to-detect bugs and, in the limit, severe data loss.

Just like GLORP, UCL-GLORP also depends on class models, table models and mappings but, in most cases, these are automatically generated. To this end, UCL-GLORP explores the *convention over configuration* design paradigm, meaning that it can take advantage of a small set of conventions in order to generate all the necessary models and mappings. For example:

– Class and slot names are used to infer database table and column names.

– Slot type declarations are used to infer the database column types and foreign keys:

  - A slot type declaration for a class belonging to the set of mapped classes is used to infer a foreign key column.

  - A slot type declaration for a type that does not belong to the set of mapped classes is used to infer an appropriate database column type.

  - A slot type declaration of the form `(or` *`type`* `null)` means that whenever the value in the database is `NULL`, the corresponding object slot has `nil` instead of being unbound.

  - A slot type declaration of the form `(vector` *`type`*`)` or `(list-of` *`type`*`)` or `(one-to-many` *`type`*`)` is used to infer a one to many relation.
  - A slot type declaration of the form `(many-to-many` *`type`*`)` is used to infer a many to many relation.

- The presence of a `primary-key` slot reader is used to infer the primary key column.
- The presence of a slot reader whose name starts with `indexed-` is used to infer which table columns should have indexes.

The use of convention over configuration is one feature of UCL-GLORP that clearly improves the creation of models and mappings that is provided by GLORP. However, the use of the conventions does not entail loosing power: the programmer can always access and modify the automatically generated configuration.

Given the fact that there is more than one strategy for mapping inheritance relations, at this moment UCL-GLORP does not attempt to infer the necessary type resolvers for mapping an inheritance tree, meaning that these type resolvers must be added manually. We plan to address this problem in the future.

4.2 Slot Access Protocol

UCL-GLORP attempts to be non-intrusive, meaning that it is possible to use plain CLOS classes to define the data model and then map those classes into database tables. One critical point of this mapping is the lazy loading of referenced objects. GLORP implements it using the proxy design pattern. UCL-GLORP implements it using the (non-meta) slot access protocol: each time an object is reconstructed from the information stored in the database, we delay the load of all its associations and the corresponding slots will remain unbound. However, the first time one of those unbound slots is accessed, we detect the unbound slot condition and we identify whether the condition represents a delayed load. In this case, we retrieve the necessary information from the database to construct the delayed object, we store it in the previously unbound slot, and we continue the computation.

This approach requires us to be prepared to handle the unbound slot condition. Obviously, we need to execute all code that potentially needs to access the database in the dynamic scope of an `handler-bind`. This is not problematic because, similarly to the manipulation of files, the managing of database connections already suggests the use of dynamic scope. What is problematic is the reaction to the unbound slot condition because the Common Lisp specification is not sufficiently clear regarding the name (or even existence) of the restart that should be used in that situation. Although one can argue that an `unbound-slot` condition is a subtype of a `cell-error` condition and these errors should have `use-value` and `store-value` restarts, the Hyperspec also includes a short note mentioning that "No functions defined in this specification are required to provide a `use-value` restart."[4]

This is an area where we think that the Common Lisp specification should have gone farther and should have specified more conditions and restarts. The hierarchy of conditions presented in the language specification is quite short and makes it difficult to develop portable programs that can handle exceptional situations. The lack of standardized restarts is also an obstacle that could have been more easily removed with a more stringent specification.

It is arguable whether treating exceptional situations as "normal" situations is an adequate approach but, given the fact that Common Lisp is one of the few languages that allow programmatic access to the condition reporting and handling mechanisms, it would be good if those mechanisms were portable across different implementations. It is not a matter of debugging convenience; it is a matter of programming convenience.

4.3 Function Introspection

Besides mapping object oriented models to relational models, GLORP also maps Smalltalk blocks to SQL statements. Similarly, as was shown in section 3, UCL-GLORP maps functions to SQL statements. To this end, it is necessary to introspect the function so that an abstract syntax tree (AST) can be built in order to rewrite it in terms of database operations.

To construct this AST, GLORP applies the predicate block to an element of a special class that does not implement any of the methods that might be called in the block but that implements the `#doesNotUnderstand:` method so that it records each method that was called, along with its arguments. It then returns another instance of the same special class to continue the construction of the AST. Certain method calls are specially recognized so that other blocks that occur in the code can also be dealt with. Obviously, there is an infinite number of Smalltalk blocks (e.g., all those that cause side-effects) where this introspection strategy cannot possibly work but, in practice, the blocks that need to be introspected are used only as predicates and, usually, these are made of boolean expressions and reader methods that do not cause any side-effects.

Porting this introspection strategy to Common Lisp was exceedingly difficult. Trying to be faithful to the Smalltalk approach, we also used an instance of a special class as predicate argument. However, instead of using the `#doesNotUnderstand:` approach that doesn't exist in Common Lisp, we used

[4] Independently of what the specification says, at least one important Common Lisp implementation didn't provide the correct restarts for the `unbound-slot` condition.

two different approaches. The first one is based on the fact that most generic function calls and, particularly, slot readers, will not be applicable to our special instance.[5] When the error is detected, we immediately define an additional method that specializes the generic function in question for our special class (so that it registers the call) and we invoke the `continue` restart so that the call is indeed registered and the introspection process can proceed.

Unfortunately, this contorted scheme cannot work with non-generic functions because it critically depends on the `continue` restart that is not generally available and, moreover, it can't detect the use of boolean operators because (1) `and` and `or` are macros that expand into special forms and (2) `not` accepts anything as argument, never signaling any error. This is where our second approach is applied: we shadow those symbols and provide different implementations so that we can have an handle on their evaluation and we also do this for all non-generic functions that might occur in a predicate that will be used for restricting a database query, such as the `some` and `string=` functions that we presented in section 3. Although it is not measurable in our experiments, we are aware that replacing (normal) functions with their generic counterparts might have a considerable impact on the performance. However, without these drastic measures, we found it highly difficult to introspect Common Lisp functions.

It should be mentioned that there are simpler ways in Common Lisp to obtain the AST of a given function. For example, in an implementation that supports it, the application of `function-lambda-expression` to a given function returns a lambda expression that can be compiled to reconstruct the function. However, this is only useful if the given function was defined in the null lexical environment. In the non-null lexical environment case, we need to access the values of the free variables but that is something that is not provided by Common Lisp, making it impossible to translate the function body to the correct SQL clause. As a result, in the general case, we still need to introspect the actual function.

5 Conclusions and Related Work

In this paper, we presented UCL-GLORP, a Common Lisp reimplementation of GLORP, an well-established ORM for Smalltalk. UCL-GLORP was developed by the author in 2005 to replace the persistency layer of a web application that previously used a very simple object-oriented database that was suffering from severe performance problems and frequent data corruptions. The entire translation process required around three months of work by the author, largely exceeding the initial estimate. In general, the delays were caused by the subtle differences between Common Lisp and Smalltalk that caused hard-to-detect bugs

[5] Unfortunately, the ANSI Common Lisp specification does not specify the subtype of error that should be signaled and all the implementations tested simply signal an instance of `error` with different error messages.

in the Common Lisp version. Without the excellent debugging tools available in the Smalltalk and Common Lisp implementations, we are convinced that the debugging phase would have taken a much longer time.

UCL-GLORP differs from GLORP in several important ways:

- UCL-GLORP infers models and mappings from a set of CLOS classes. This is something that is beyond GLORP capabilities because, contrary to CLOS, Smalltalk classes do not have any standardized way of annotating slots with the necessary type information.
- UCL-GLORP never expose proxies. Instead, these are completely hidden from application code and are resolved whenever we trap the `unbound-slot` condition associated with the corresponding slot access. This is a much safer approach to lazy loading because, contrary to GLORP, it is impossible, in UCL-GLORP, to create identity problems between a proxy and the object it stands for.
- UCL-GLORP is more complex than GLORP because we need to deal with a lot more diversity in Common Lisp than in Smalltalk. One of the strongest points of Smalltalk is, indeed, its simplicity and uniformity that makes it easier to centralize behavior.
- There is no support in GLORP for schema evolution. UCL-GLORP, on the contrary, provides such support. Besides mapping a set of classes into a set of database tables, UCL-GLORP is also capable of mapping a set of class *changes* into a set of database *changes*. Sometimes, there is more than one way to do this and whenever this happens, UCL-GLORP presents the different options and requests guidance from the programmer.

One of the main responsibilities of an ORM is to ensure the persistence of data. In the Common Lisp camp, this task can also be accomplished using any of the other persistence frameworks, namely, UCL+P [JS97], Statice [WFGL88], PCLOS [Pae88], PLOB! [Kir95], AllegroCache [Aas05], Common SQL[Lev02], CLSQL[Ros07], YstokSQL[Iva08], Postmodern[Hav08], Elephant[LB08], etc.

Besides ensuring persistence, an ORM also ensures that persistent data is stored according to the principles of the relational model, a requirement that cannot be satisfied by some of the previous Common Lisp frameworks: UCL+P and PLOB! use a persistent heap; Statice, PCLOS, Elephant and AllegroCache are object-oriented databases.

Finally, the most important responsibility of an ORM is to provide a mapping from the object-oriented model to the relational model and this is the requirement that excludes almost all Common Lisp frameworks. Common SQL and CLSQL map objects into relational tables but, as explained in the introduction, they do not provide all the necessary features; YstokSQL only provides a Lisp-like SQL syntax; and Postmodern, although it provides some mapping capabilities, is not intended as a full-fledged object-relational system.

CLOS-DB [Wer08] is another proposal for mapping objects into tables but it requires the user to explicitly provide functions (and SQL strings for the less trivial cases) that update the database whenever the objects change and it also requires explicit SQL code to manually populate the database. Gestalt-class [MHL96] is a more sophisticated system that CLOSifies GESTALT [HN89]—an interface to multiple heterogeneous database systems. Gestalt-class maps a GESTALT database schema into automatically generated CLOS classes and it provides some of the features of a proper ORM, such as preserving the identity of the loaded objects and mapping S-expressions into GESTALT queries. However, GESTALT is not a relational database, although it can use a relational database as a backend. When this happens, the relational model used is fixed: each class is associated with two primary tables and zero or more association tables. The primary tables are used to store all single-valued attributes (value objects and one-to-one relations) and the next available *object id*; the association tables are used to store the one-to-many relations. This is similar to what UCL-GLORP does, although we prefer database sequences for the *object id*, we have a more natural mapping for one-to-many relations that does not require an additional association table and we can also use many-to-many relations. Besides these small differences, there is a huge difference in the approach used: UCL-GLORP can infer the relational model from the object-oriented model while Gestalt-class operates in the opposite direction: the programmer defines attributes and types (sets of attributes) and Gestalt-class infers the CLOS classes that represent them.

Very recently, another ORM for Common Lisp was presented: CL-PEREC [LMB08]. Although it targets the same goals, CL-PEREC and UCL-GLORP have several important differences:

- In CL-PEREC, classes whose instances should be persistent must belong to a special metaclass. There is no such requirement in UCL-GLORP and plain CLOS instances can be made persistent without any changes.
- CL-PEREC does not include relations in the class definitions. Instead, all relations must be defined separately. UCL-GLORP, on the other hand, can infer the relations from the class definitions.
- CL-PEREC provides a specialized SQL-like query language that can be compiled (macro-expanded, actually) to a more efficient form where some parts are computed in advance. Although we didn't mentioned it in this article, UCL-GLORP also provides an SQL-like language but this language is not generally used by the programmer. Instead, it is used as the target for the translation of Common Lisp functions that restrict the queries.

– Contrary to UCL-GLORP, CL-PEREC doesn't implement units of work and transactions are not supported on the object level. This means that every slot update is immediately transferred to the database, thus preventing the optimizations and reorderings that are done by UCL-GLORP.

Besides the mentioned differences, there is a more profound mismatch between CL-PEREC and UCL-GLORP: CL-PEREC provides a new language for class definitions and queries while UCL-GLORP tries very hard to remain faithful to the "normal" CLOS style. We think we achieved this goal because, using UCL-GLORP, programs using CLOS can be made persistent without requiring incompatible changes. This is an important property because it makes the program independent of the persistency backend used.

Being non-intrusive is a fundamental goal for UCL-GLORP. However, there is always a balance between hiding the implementation details and achieving good performance: UCL-GLORP allows us to provide transparent object-relational persistency to Common Lisp programs but the performance consequences might be more difficult to predict.

As we explained in this paper, in order to improve the performance we might have to rewrite parts of the code. One partial solution to this problem is to provide a set of compiler macros that automatically rewrite the common use cases but, in general, this is a problem that will always require human intervention. In the limit, for critical optimizations the solution might be to manually provide the models and mappings and to explore the low-level UCL-GLORP SQL interface, using a more persistency-aware development model.

Although there are still several rough edges that we would like to smooth, UCL-GLORP is perfectly usable and, in fact, we have been using UCL-GLORP in a production environment for more than two years, to provide the persistency layer of a web-based application.

On the negative side, it should be mentioned that portability is UCL-GLORP major problem because it stresses Common Lisp in ways that are not well-defined in the language specification. Although not strictly related to the features provided by UCL-GLORP, the code base also explores a few Common Lisp features that are not provided by all Common Lisp implementations. For example, we depend on CLOS introspection at macro-expansion time, a possibility that is explicitly allowed by the standard for macros that care to provide the compile-time environment. Unfortunately, at the time of this writing, some of the free Common Lisp implementations, including SBCL and CMUCL, do not allow that, thus preventing UCL-GLORP of working in those implementations. The net result of all these limitations is that, at the moment, UCL-GLORP only runs in Allegro Common Lisp and Lispworks, but we expect that the situation will improve in the near future.

Also on the negative side is the fact that large parts of the code base are written in a non-standard style that still contains many traces of the original implementation language. This was necessary to speed up the translation process but it makes it difficult for others to correct bugs and/or implement new features. We plan to address this problem by rewriting the problematic parts according to the Common Lisp usual programming conventions and then we will make the source code freely available.

We will continue to develop UCL-GLORP in order to address some of the shortcomings discussed in this paper. On the top of our list is the automatic creation of mappings for dealing with class hierarchies, where we plan to generate a combination of filtered mappings, for shallow hierarchies, with horizontal mappings for the other cases.

References

[Aas05] Jans Aasman. AllegroCache: A high-performance object database for large complex problems. In *5th International Lisp Conference*, Stanford University, June 2005.

[Fow05] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison Wesley, 2005.

[GHJV00] Gamma, Helm, Johnson, and Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 2000.

[Hav08] Marijn Haverbeke. Postmodern. `http://common-lisp.net/project/postmodern/`, Nov 2008.

[HN89] Michael L. Heytens and Rishiyur S. Nikhil. Gestalt: an expressive database programming system. *SIGMOD Rec.*, 18(1):54–67, 1989.

[Inc00] Franz Incorporated. LLGPL, the Lisp Lesser GNU Public License. `http://opensource.franz.com/preamble.html`, 2000.

[Iva08] Dmitriy Ivanov. YstokSQL. `http://lisp.ystok.ru/ysql/`, Nov 2008.

[JS97] J. H. Jacobs and Mark R. Swanson. UCL+P - defining and implementing persistent common lisp. *Lisp and Symbolic Computation*, 10(1):5–38, 1997.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kir95] Heiko Kirschke. Persistency in a Dynamic Object-Oriented Programming Language. Technical Report 10, University of Hamburg Computer Science Department, Jul 1995.

[Kni00] Alan Knight. Glorp: generic lightweight object-relational persistence. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 173–174, New York, NY, USA, 2000. ACM.

[LB08] Ben Lee and Andrew Blumberg. Elephant. `http://common-lisp.net/project/elephant/`, Nov 2008.

[Lev02] Nick Levine. Common SQL. `http://www.ravenbrook.com/doc/2002/09/13/common-sql/`, September 2002.

[Lie89] K. J. Lienberherr. Formulations and benefits of the law of demeter. *SIGPLAN Not.*, 24(3):67–78, 1989.

[LMB08] Attila Lendvai, Levente Mészáros, and Tamás Borbély. cl-perec: RDBMS based CLOS persistency. `http://common-lisp.net/project/cl-perec/`, Feb 2008.

[MHL96] Michael B. McIlrath, Michael L. Heytens, and Thomas J. Lohman. Gestalt-class: A Persistent, Multi-user CLOS Application Environment. In *Dynamic Objects Workshop*, 1996.

[Pae88] Andreas Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming.* Lecture Notes in Computer Science, Springer Verlag, 1988.

[Ros07] Kevin M. Rosenberg. CLSQL – a multi-platform SQL interface for Common Lisp. `http://clsql.b9.com/`, September 2007.

[Wer08] Christoph Wernhard. CLOS-DB. `http://cs.christophwernhard.com/cdb/`, Nov 2008.

[WFGL88] D. Weinreb, N. Feinberg, D. Gerson, and C. Lamb. An object-oriented database system to support an integrated programming environment. *Data Engineering*, 11(2):33–43, June 1988.