Between Types and Tables
Using Generic Programming for Automated Mapping Between Data Types and Relational Databases

Bas Lijnse and Rinus Plasmeijer

Radboud University Nijmegen
{b.lijnse,rinus}@cs.ru.nl

Abstract. In today's digital society, information systems play an important role in many organizations. While their construction is a well understood software engineering process, it still requires much engineering effort. The de facto storage mechanism in information systems is the relational database. Although the representation of data in these databases is optimized for efficient storage, it is less suitable for use in the software components that manipulate the data. Therefore, much of the construction of an information system consists of programming translations between the database and a more convenient representation in the software.

In this paper we present an approach which automates this work for data entry applications, by providing generic versions of the elementary CRUD (Create, Read, Update, Delete) operations. In the spirit of model based development we use Object Role Models, which are normally used to design databases, to derive not only a database, but also a set of data types in Clean to hold data during manipulation. These types represent all information related to a conceptual entity as a single value, and contain enough information about the database to enable automatic mapping. For data entry applications this means that all database operations can be handled by a single generic function.

To illustrate the viability of our approach, a prototype library, which performs this mapping, and an example information system have been implemented.

Keywords: Generic programming, Functional programming, Clean, Relational databases, Object role models, Model based software development

1 Introduction

In today's digital society, information systems play an important role in many organizations. Many administrative business processes are supported by these systems, while others have even been entirely automated. While the construction of such systems has become a more or less standardized software engineering process, the required amount of effort remains high. Because each organisation

has different business processes, information systems need to be tailored or custom made for each individual organisation.

One of the primary functions of information systems is to create, manipulate and view (large) persistent shared collections of data. The de facto storage mechanism for these data structures is the relational database, in which all information is represented in tables with records that reference other records. Although this representation is optimized for redundancy free storage of data, it is less suited for direct manipulation of that data. The reason for this is that conceptually elementary units are often split up into multiple database records. For example, in a small business system, a project consisting of a name and a number of tasks is broken down into one record for the project and a number of records for the tasks which each reference the project.

In data entry applications it is more convenient for developers to do operations on conceptual units instead of single database records. To reuse the example, adding a project instead of adding a project record and a number of task records. Therefore, in the programming language we use to build the data entry components, we need data structures that represent conceptual units rather than database records. While it is easy to construct a type in most modern languages to represent a conceptual unit as a single data structure, using any type more complex than a single database record means that some translation is required whenever data enters or leaves the database. As a result, since each system has a unique database design, a lot of boiler plate code has to be be written to achieve this translation. This translation code is all very similar except for the types and tables they translate between. Even when a DSEL is used to abstract the database interaction from low level SQL, one still has to define the mapping for each new type. This repetitive programming work is not only mind numbing for developers, it is also time consuming and error-prone. Over the years several tools and libraries have been developed to solve this issue with varying degrees of success and practical use. We discuss these approaches in detail in section 6.

In this paper we present a novel approach based on generic programming in Clean that provides generic versions of the elementary CRUD (Create, Read, Update, Delete) operations that abstract over types and tables. These operations map changes in data structures that reflect the conceptual unit structure of entities, to changes in a relational database. The main prerequisite for enabling this, is that all necessary information about the entities' database representations can be inferred from the types of these data structures. In the spirit of model based development, we do this by deriving both the data types and a relational database from the same high level data model. The language we use for these models is Object Role Modeling (ORM). In this graphic modelling language one can specify what information is to be stored in an information system by expressing facts about the modelled domain. Since ORM has a formally defined syntax and semantics, it enables the derivation of a set of database tables, as done in the standard Rmap algorithm [10], or a set of types in our approach.

Our approach consists of four mappings between representations on different levels of abstraction that are depicted in Fig. 1. The first step (1) is a mapping

from ORM models on a conceptual level to a set of Clean types on the type level. From these types we derive a matching relational model for storage in a database (2). Our generic library is then used at the value level (3) to do CRUD operations on values of the representations where it automatically maps the values to the database. For many existing databases it is also possible to reverse engineer a set of representation types from a relational model (4).

The key idea behind our approach is that it addresses the representations of data for storage and manipulation as two sides of the same coin. Instead of focusing on either using databases as storage for Clean values, or on Clean values as interface to a storage representation, we consider Clean values and databases as different representations of the same high-level concepts.

Although our approach involves many stages of the software engineering process, we consider the following to be the main contributions of this paper:

– We introduce a structured method to derive Clean data types from ORM models, that allow capturing all information about a conceptual entity in a single data structure. The details of this process can be found in section 3.
– We present a generic library which provides the four CRUD operations as functions on single data structures. These operations also work when the representations in the database span multiple tables. Especially in data entry applications, this library can replace much boiler plate code. The CRUD operations are covered in section 4 and the implementation of the library is discussed in section 5.
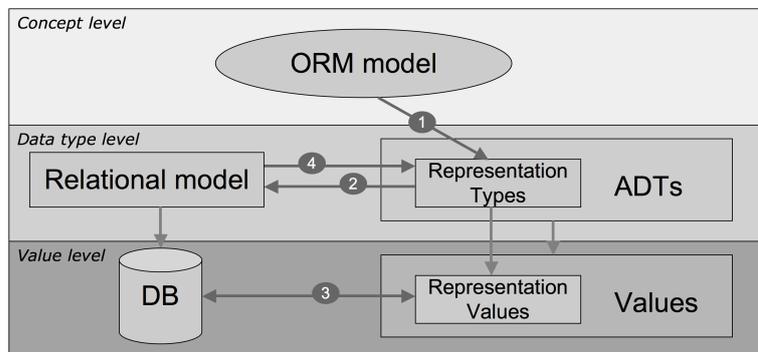

Fig. 1. The four steps in our method.

2   Motivating Example

To illustrate the various steps in our approach, and to provide some feeling about how it can be applied, we will make use of a running example throughout the

remainder of this paper. This example is a simple project management system for a typical small business, which stores information about the following conceptual entities:

- **Projects** are abstract entities which are identified by a unique project number and have a textual description. Projects are containers for tasks and can be worked on by employees. A project can be a sub project of another project and can have sub projects of its own.
- **Tasks** are units of work that have to be done for a certain project. They are identified by a unique task number and have a textual description. The system also keeps track of whether a task is finished or not.
- **Employees** are workers that are identified by a unique name and have a description. They can be assigned to work on projects. An employee can work on several projects at a time and multiple employees may work on the same project.

2.1 ORM Formalization

To enable our generic mapping we need to make the above specification more precise. Using ORM [5], we can make a formal conceptual model of the example as shown in Fig. 2. Using ORM, one models *facts* about *objects*. Facts are expressed as semi-natural language sentences. For example: "**Employee** a *works on* **Project** b". An ORM model abstracts over concrete facts about concrete objects by defining *fact types* (the boxes) and *object types* (the circles). Unlike other data modeling languages like ER [3] or UML[14], ORM does not differentiate between relations and attributes, but considers only facts. ORM also models several basic constraints on the roles that objects have in facts. One can express uniqueness, meaning that a fact about some combination of objects occurs at most once, and mandatory role constraints which enforce that a fact about a certain object must occur at least once. In Fig. 2, these constraint are depicted as arrows spanning unique combinations of roles, and dots on roles that are mandatory.

3 Types and Tables

The key idea on which our approach is based is that, in data entry applications, we want to manipulate single data structures that represent a conceptual unit. What we do not want, is to manually specify the queries required to build such data structures, or to specify how to update the database after the data structure has been altered. Unfortunately this is often necessary because, since types in data entry applications are often defined ad-hoc for a separately designed database, the relation between types and tables is unclear and inconsistent.

We improve this situation by using a structured process. Since a relational database, and the Clean types used for manipulating it, are simply two different representations of the same abstract entities, the obvious thing to do is define
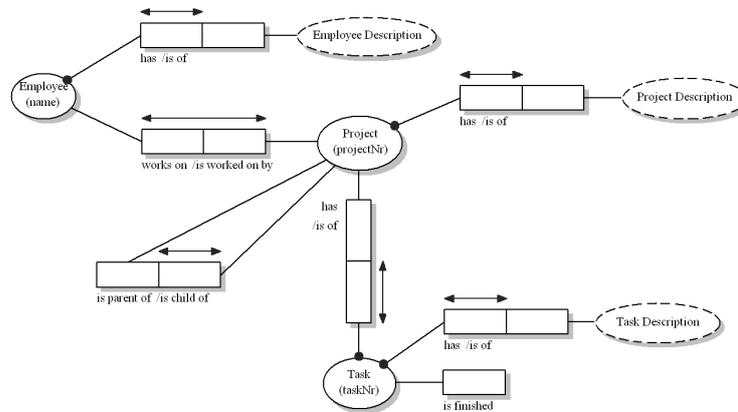
Fig. 2. A simple ORM model for a project management system

a high level specification of these abstract entities and use it to derive both representations. In the next section we show that when enough information about the storage representation of objects can be inferred from the types of their corresponding Clean representation, we can define an automated mapping once and for all using generic functions. In this section we show how we can obtain a set of types and tables for which this property holds.

3.1 Object Role Models

Instead of defining our own language for defining conceptual entities, we use an existing language from the information modeling field: Object Role Modeling. However, for reasons of simplicity, our approach only considers ORM models that satisfy the following constraints:

- The model only contains entity types, value types and unary and binary fact types.
- Each entity type can be identified by a single value.
- Uniqueness constraints on single facts and mandatory role constraints are the only constraints used.
- Each fact type has at least one uniqueness constraint.
- Uniqueness constraints spanning two roles are only used for facts concerning two entity types.

Although this subset of ORM neglects some advanced ORM constructs, like subtyping or n-ary fact types, it has roughly the same expressive power as the widely used Entity Relationship (ER) [3] modeling language, and is sufficient for most common information systems. Nonetheless, we still use ORM instead of ER because it allows extension of our method to even more expressive conceptual models in the future.

3.2   Representation Types

Although a solid conceptual model is the basis of a well-designed information system, from a programmers perspective however, we are more interested in the concrete representation as types in our (Clean) applications.

Conceptual entities can have different types of relations and constraints. When we want to represent conceptual objects as single Clean data structures we need types that can contain all facts about an entity and also retain information about constraints and relations. This is achieved by defining a subset of Clean's record types with meaningful field names. This set is defined as follows:

- **Entity Records**
  Clean records are used as the primary construct to represent conceptual entities. These records have the same name as the entity type they represent, and have fields for every fact type concerning an entity. The names of these fields have a mandatory structure which can have the following three forms:
  - `<entity name>_<value name>`
    This form is used for values or entities that have a *one-to-one relationship* with this entity. The entity name is a unique name for this entity type, typically the same as the name of the record type. The first field of an entity record must always have this form and is assumed to be a unique identifier for the current entity.
  - `<entity name>_ofwhich_<match name>`
    This form is used for embedding relations between two entities where the relation between the two entities is defined such that the value of the match name of one of the entities is equal to the identity value of another entity. This form is used for *one-to-many relations* between entities. The entity name is the identifier of the "many" part of the relationship. The current entity is the "one" side of the relation.
  - `<relation name>_<select name>_ofwhich_<match name>`
    This form is used for *many-to-many relationships* between entity types. The relation name is a unique name for this relation and is used by both entity records that have a role in the relation. The select and match names are role identifiers for both parts of the relation.

  The types that fields in an entity record are allowed to have, are limited as well. They can be of scalar type, another entity or identification record type, or `Maybe` or list of scalar or entity or identification record type.
- **Identification Records**
  Because we do not always want to store or load an entire database, we need a representation for references to entities that stay in the database. We represent these references as identification records. These are records that have the same name as the entity record they identify, with an "`ID`" suffix. These records contain exactly one field which has the same name and type as the corresponding entity record.
- **Scalar Types**
  Value types in ORM are mapped to the basic scalar types in Clean: `Int`, `Bool`, `Char`, `String` and `Real`.

- **List and Maybe types**

  When the uniqueness and total role constraints on a fact type define that a fact can be optional, or can have multiple instances, we use Clean's list (`[a]`) and `Maybe` (`::Maybe a = Nothing | Just a`) type to wrap the type of the object involved in the fact. It is important to note that the order of lists is considered to have no meaning in these types. Storage of an entity record which contains a list does therefore not guarantee that this list has the same order when read again.

Using these types, the ORM model of our project management system (Fig. 2) can be represented by the set of Clean types given below.

```
:: Employee =   { employee_name                          :: String
                , employee_description                    :: String
                , projectworkers_project_ofwhich_employee :: [ProjectID]
                }
:: EmployeeID = { employee_name                           :: String
                }
:: Project =    { project_projectNr                       :: Int
                , project_description                      :: String
                , project_parent                          :: (Maybe ProjectID)
                , task_ofwhich_project                     :: [Task]
                , project_ofwhich_parent                   :: [ProjectID]
                , projectworkers_employee_ofwhich_project  :: [EmployeeID]
                }
:: ProjectID =  { project_projectNr                       :: Int
                }
:: Task =       { task_taskNr                              :: Int
                , task_project                             :: ProjectID
                , task_description                         :: String
                , task_done                                :: Bool
                }
:: TaskID =     { task_taskNr                              :: Int
                }
```

An interesting property of these types is that, unlike database records these Clean records can also contain nested representations of related objects.

3.3 From ORM To Representation Types

To make sure that a set of representation types represent the right concepts, we systematically derive the types from an ORM model (mapping (1) in Fig. 1). The algorithm to perform this mapping groups fact types in a similar fashion as the standard Rmap [10] algorithm and is summarized below. A more elaborate description can be found in [8].

1. For each entity type in the ORM, define an entity and identification record in Clean. They both have one field, which will have the name and type of the primary identification of the entity in ORM.

2. Add fields to the entity records. Each entity record will get a field for all the fact types in which it plays a role. The types and names of the fields are determined based on the object types and constraints in the model.
   – When the entity type is related to another entity type, the type of the field is the identification record for that entity. When it is related to a value type, the field will have a scalar value. The name of the field may be freely chosen but has to be prefixed with a globally unique entity identifier. The obvious choice for this is the name of the entity type.
   – When the fact type is unary, the field's type will be `Bool`.
   – When there is no mandatory role constraint on the role an entity is playing, the field's type will be a `Maybe` type.
   – When there is no uniqueness constraint on the role an entity is playing the field's type be a list type.
   – Each field name is prefixed with a grouping identifier. If a fact type can be attributed completely to the entity we are defining the type for, we use the name of the entity as prefix. If not, we choose a unique prefix for that fact type, that is to be used in the entity records of both entities playing a role in the fact type.
3. Optionally replace identification record types in record fields to entity record types. This allows the direct embedding of related entities in the data structure of an entity. One has to be careful however to not introduce "inclusion cycles". When an included related entity embeds the original entity again, a cycle exists which will cause endless recursion during execution.

Because step 3. is optional, and the choice between inclusion or reference depends on the intended use of the representation types, this transformation can only be automated by an interactive process or annotation of the ORM model.

3.4 From Representation Types to Tables

The next step in our approach is getting from a set of representation types to a relational model (mapping (2) in Fig. 1). The obvious way would be to map from ORM directly to a relational model as is done in the standard Rmap algorithm [10]. However, since the representation types are already very close to the relational structure, it is easier to derive the tables from these types. A summary of the mapping process is given below. A more detailed version can be found in [8].

1. Define tables for all entities. In these tables all record fields are grouped that have the same entity name as the first (identification) field of the record. The types of the columns are the types of the record fields in the case of scalar types. In the case of entity or identification records the column gets the type of the first field of these record types. When a record field's type is a `Maybe` type, the corresponding column is allowed to have `NULL` values.
2. Define tables for all many-to-many relations. For all many-to-many relationships find the pairs of relation names and define a two-column table by that

name. The names of the two columns are the entity names found in the record fields in the representation types.

3. Add foreign key constraints. Everywhere an entity or identification type in the record field is mapped to a column in a table, a foreign key constraint is defined that references the primary key of the table of the corresponding entity type.

When this algorithm is applied to the set of representation types of section 3.2, we get the set of database tables depicted in Fig. 3. Since this algorithm is completely deterministic it can be easily automated.
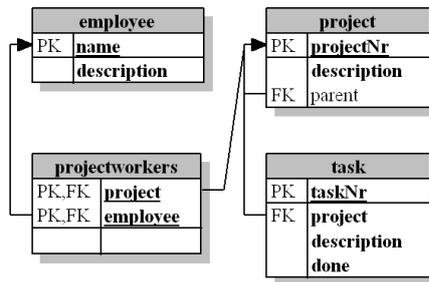

Fig. 3. The derived tables of the ORM model in Fig. 2

With this mapping, we have done all the preparatory work that is required to use our generic library. For new information systems, this is all the initial work one has to do: Define an ORM model, derive a set of representation types and derive a relational model from those types.

3.5 Reverse engineering: From Tables to Representation Types

In situations where we already have a database, that we want to interface with, we still want to be able to use our generic library. In many situations we are able to reverse engineer a set of representation types from an existing relational model to make this possible.

The process itself (mapping (4) in Fig. 1) is a rather trivial inverse operation of the method to derive a relational model from the representation types. However, this is only possible under certain conditions:

– The relational model must only contain tables indexed on a single column primary key that represent entities and two column tables with a primary key spanning both columns that represent additional facts. When this condition holds, there exists a set of representation types from which we could have derived the existing database.

– We must know which columns are used as references, and what entities they reference. Since the use of foreign keys is not obligatory, it is not always possible to infer the references in a relational model. We can only define a set of representation types if we know which conceptual entities are related and how.

When these conditions hold, which often do for simple information systems, we are able to use our library even in situations where no ORM model of the existing system is available. When these conditions do not hold for the complete database, but do hold for a part of the database, it is still possible to define a set of types for that part. Such partial use of the generic mapping, can still save a lot of work.

4 Generic CRUD Operations

Although having Clean types and database tables that have a clear relation with a formal conceptual model is a merit on its own, the point of that exercise was to enable generic CRUD operations.

What we want to achieve is a set of four generic functions that are available for every possible representation type and enable us to manipulate the entities in the databases of our information systems. Ideally the type definitions of this set would look somewhat like the following code:

```
create :: entity db → (ref,    db)
read   :: ref    db → (entity, db)
update :: entity db →        db
delete :: ref    db →        db
```

Here `ref`, `entity` and `db` are type variables for respectively the identification record type, the entity record type and a database cursor type. Obviously it is not possible to create such a completely polymorphic set of functions, but we can come very close using generics and overloading.

In this section we show how two of these operations, read and update, work by means of an example. The other two are similar and are not covered for the sake of brevity. A full detailed description of all four operations can be found in [8]. In the example we will assume the conceptual model of Fig. 2, the types of section 3.2, and the database tables of Fig. 3.

4.1 Reading objects

The first operation we will show is the generic read. Suppose we have a database with the following information about some project:

– It has projectNr 84, description "Spring brochure" and the project has no parent project and no sub projects.
– A task is defined for this project with taskNr 481 and description "Draft text" which is not done yet.

- Another task is defined with taskNr 487 and description "Call printer about price" which is also not done yet.
- Employees "john" and "bob" are working on this project.

All of this information can be read into a single Clean value of type `Project` in just one line of code[1]:

```
(mbError, mbProject, cur) = gsql_read {ProjectID|project_projectNr = 84} cur
```

If all goes well, this will give us the following data structure:

```
{ Project | project_projectNr = 84
  , project_description = ''Spring brochure''
  , project_parent = Nothing
  , task_ofwhich_project
  = [ { Task | task_taskNr = 481, task_project = 84
      , task_description = ''Draft text'', task_done = False
      }
    , { Task | task_taskNr = 487, task_project = 84
      , task_descrption = ''Call printer about price'', task_done = False
      } ]
  , project_ofwhich_parent = []
  , projectworkers_employee_ofwhich_project
  = [ {EmployeeID | employee_name = ''john''}
    , {EmployeeID | employee_name = ''bob'' } ]
}
```

This single line of code gives us a lot for free. If we had to write a `read_product` function by hand, it would have required three different SQL queries plus a conversion from flat lists of SQL data values to the nested `Project` structure.

To achieve this generically, two problems have to be solved: 1. How do we find the information in the database? And 2. how do we construct a value, in this case of type `Project`? The first problem is solved by interpreting the field names of record types and translating them to SQL queries. The results of these queries are then systematically concatenated to produce a stream of values (tokens) which is a serialized representation of the value we want to construct. This reduces the second problem to deserialization of that representation.

Instead of describing the read operation at an abstract level, it is easier to see what happens by following it step by step when used to read the `Project` described above.

1. The first step we take is serialization of the `ProjecID` value to create an initial token stream. Thus in this case, the read operation is started with initial stream [84][2].
2. The next step is to apply the instantiation of the generic read operation for the `Project` type. When the read operation is applied to read an entity

---

[1] In this code the variable `cur` is a unique database cursor used to query the database.
[2] To illustrate the intermediate values of the token stream we use an ad-hoc untyped list notation. This is **not** Clean syntax.

record, the first thing that is done is to expand the token stream by reading additional data for all fields of the record. The head of the token stream is used to match database records and the SQL queries are constructed from the information encoded in the field names. For example, the data for the field `project_description` is retrieved with the SQL query: `SELECT description FROM project WHERE projectNr = 84`. When an optional field is empty a NULL token is added to the stream and when a field has multiple values a terminator (TRM) token is added after the last value.

So for the example project, the token stream has the value after expansion: [84, "Spring brochure", NULL, 481, 487, TRM, TRM, "john", "bob", TRM]

3. With the data for all project fields read, the read operation is applied recursively to construct the record fields. When the read operation is instantiated for basic types or identification records no additional data is read. Instead, tokens are consumed to construct values. So after the values of the first three fields (84, ``Spring brochure'' and `Nothing`) are constructed the token stream has the value: [481, 487, TRM, TRM, "john", "bob", TRM]

4. The instantiation of the read operation for lists will repeatedly apply the read operation for its element type until the head of the token stream is a terminator. So in this case, the create operation for type `Task` will be called twice. Because `Task`, like `Project`, is an entity record type, we read additional data again. After expansion of the first task the stream has value: [481, 84, "Draft text", false, 487, TRM, TRM, "john", "bob", TRM]

When the list of both tasks is read and constructed the stream is reduced to: [TRM, "john", "bob", TRM]

5. Thus the process continues, and when recursion is completed for all fields we have an empty token stream and can construct the `Project` record.

4.2 Local changes with global meaning

Once all facts about an object are read into a Clean data structure, we can change it in a program. Because this structure is not just some convenient grouping of values for computation, but has a meaningful relationship with both the underlying conceptual model and the relational model in the database, we can interpret changes to this data structure as changes on the conceptual level.

To illustrate this we make some changes to the example `Project` of the previous section and consider their meaning on the conceptual level.

- We change the value of the `project_description` field to ``Summer brochure''. The meaning of this change is simple. Since each project has exactly one description, this new description will replace the old value in the database.
- We change the value of the field `task_done` of the first `Task` in the list to `True`. The meaning of this change is simple as well. Since each task is either done or not, this new value will replace the value in the database. So although the task is embedded in the project value, it is still a separate object on the conceptual level which facts can be changed.

– We remove the second `Task` from the list.
  The meaning of this change is less obvious. Since tasks and projects are both conceptual objects that happen to be related, does a removal from the list mean that the conceptual task object and all its facts are removed? Or does it mean that just the relation between the task and project is removed? For the representation types, this choice is dependent on the used type. For entity records, like `Task`, we will interpret removal of the list as complete removal of the object. For identification records, like `TaskID`, we will only remove the relation between objects. Thus in this case task 487 will be deleted completely.
– We add a new `Task` defined as:

```
{ task_taskNr = 0, task_project = {ProjectID | project_projectNr = 0}
, task_description = ''Check online prices'', task_done = False
}
```

  This change means that a new task for this project has to be created. The interesting parts however are the `task_taskNr` and `task_project` fields. Each task is related to exactly one project. We have specified in the task record that this is project 0. But this task is created as part of the list of tasks of project 84. When new objects are created in the context of another object we will let the context take precedence and ignore the specified identification. Hence, this change means that a new task is created which is related to project 84, not 0.
  The `task_taskNr` field is also interesting. For the identification of new objects we interpret the specified value (0) as a suggestion, but leave it up to the database to determine the actual value. This enables the use of auto incrementing counters which are commonly used in databases.
– We remove ``john'' from the list in `projectworkers_employee_ofwhich_project`.
  Because the `projectworkers_employee_ofwhich_project` field is a list of identification records, we will interpret the removal of ``john'' from this list as "john no longer works on this project" and not as complete removal of the employee named "john" from the database.

4.3   Updating objects

In the previous section we have made quite a few changes to our local representation of the project, but all of these changes can be applied to the global representation in the database at once with just the following single line of Clean code:

```
(mbError, mbProjectId, cur) = gsql_update project cur
```

This single line saves us even more programming work than the generic read function. To apply all the changes by hand would in this case require six cusom crafted SQL queries and the necessary conversion code.

As with the read operation, we illustrate the generic update by following its operation step by step.

1. The update operation for entity records is done in three recursive passes. In the first pass we consider only the fields that are single basic values or identity records. In this case the fields that start with `project_`. The update operation or basic values and identification records does no database interaction, but just serializes values to produce the token stream. After this first pass the token stream has the value: [84, "Summer brochure", NULL].

2. After this pass we update the database record for this project. Because new objects can be added (like the new task) we verify that the update query did indeed modify a record in the database. If not, we create a new record. After this update/create we know the definitive identification of this project (84) and are ready for the next pass.

3. In the second pass we will do a recursive update of the remaining record fields. To make sure that the identification context object takes precedence when updating nested objects we pass along special override tokens (OVR) that specify for which fields in the nested entity records the context must be used instead of its value. In this case the second pass is started with token stream: [OVR task_project ⇒ 84, OVR projectworkers_project ⇒ 84]. The override tokens are used during serialization in the first update pass of a nested entity record. When the second pass finishes the resulting token stream has value: [481, 532, TRM, TRM, "bob", TRM]. The value 532 is an automatically assigned identification for the newly created task.

4. In the third and final pass, the token stream of the second pass is compared with the token stream that a (non-recursive) read operation is for this project produces to determine which list elements have been removed. For these values, the generic delete operation is used to remove them from them from the database.

5. After these three passes, the identification value of the current record is added to the token stream it was started with. In this case returning a token stream of value: [84].

6. The final step is to deserialize the token stream to produce a `ProjectID` value.

4.4   Shared consequences

An interesting property of the previously illustrated generic operations is that changes in one object have consequences for related objects. Because facts are conceptually shared between objects, the operations maintain that shared structure in the database. If we would have read the `Employee` record of ``john'' before going through the example, the list in the `projectworkers_project_ofwhich_employee` would have contained the value {`ProjectID|project_projectNr=84`}. If we would read it again after updating the project, this value would no longer occur in the list.

5   Implementation in Clean

To validate the generic operations, we have implemented the operations described in the previous section as a prototype library in Clean called "GenSQL".

This library contains about 950 lines of Clean code of which roughly 500 are used for the definition of the main generic function. The rest constitutes about fifty helper functions. Because of its large size, it is not possible to present the generic function in detail. The design of the library as a whole is therefore presented instead[3].

5.1 Jack of All Trades

Because the generics mechanism in Clean has some limitations, the implementation of the operations in the GenSQL library has a somewhat unusual design. In Clean it is not possible to call other generic functions of unknown type in the definition of a generic function. The different CRUD operations however, do have some overlap in their functionality. The update operation, for instance, uses the delete operation during a garbage collect step. Because of the limitation we are not able to isolate this overlap in a separate generic function.

To deal with this limitation of the generics mechanism, all operations have been combined into one "Jack of all trades" function. The type signature of this function, `gSQL`, is as follows:

```
generic gSQL t ::
  GSQLMode GSQLPass (Maybe t) [GSQLFieldInfo] [GSQLToken] *cur   →
  ((Maybe GSQLError), (Maybe t),[GSQLFieldInfo],[GSQLToken],*cur) |SQLCursor cur
```

The first two arguments of this function are the mode and pass of the operation we want `gSQL` to perform. The mode is one of the four operations `GSQLRead`, `GSQLCreate`, `GSQLUpdate`, `GSQLDelete`, the type information mode `GSQLInfo` or `GSQLInit`. The latter serializes a reference value to the token list in order to start a read or delete operation. The `GSQLPass` type is simply a synonym for `Int`.

The next three arguments are the data structures on which the `gSQL` function operates. All three are both input and output parameters and depending on the mode, are either produced or consumed. The first argument is an optional value of type `t`. During the read and delete operations, this argument is `Nothing` in the input and `Just` in the output because values are constructed from the token list. During the create, update, info and init operations, the argument is `Just` in the input because values are serialized to the token or info list. The second argument is the token list to which data structures are serialized. The third argument is the info list. In this list, type information about record fields is accumulated. The last argument of the `gSQL` function is a unique database cursor which has to be in the `SQLCursor` type class[4]. This is a handle which is used to interact with the database. The return type of the `gSQL` function is a tuple which contains an optional error an optional value of type `t`, the token list, the info list and the database cursor.

---

[3] Full sources of both the library and the demo application can be found at: http://www.st.cs.ru.nl/papers/2009/gensql-prototype.tgz

[4] A | in a type signature is Clean notation for specifying class constraints

Although this "Jack of all trades" function is large, it is clearly divided into separate cases for the different types and modes to keep it readable and maintainable.

5.2 Convenient wrappers

Because of the all-in-one design of the `gSQL` function, it is not very practical to use. For the read and delete operations, it even has to be called twice. First in the init mode to prepare the token list, and then in the read or delete mode to do the actual work.

To hide all of this nastiness from the programmer, the GenSQL library provides wrapper functions for each of the four operations. These wrappers have the following type signature.

```
gsql_read   :: a *cur → (Maybe GSQLError, Maybe b, *cur)
    | gSQL{|*|} a & gSQL{|*|} b & SQLCursor cur
gsql_create :: b *cur → (Maybe GSQLError, Maybe a, *cur)
    | gSQL{|*|} a & gSQL{|*|} b & SQLCursor cur
gsql_update :: b *cur → (Maybe GSQLError, Maybe a, *cur)
    | gSQL{|*|} a & gSQL{|*|} b & SQLCursor cur
gsql_delete :: a *cur → (Maybe GSQLError, Maybe b, *cur)
    | gSQL{|*|} a & gSQL{|*|} b & SQLCursor cur
```

Thanks to Clean's overloading mechanism we can use these wrapper functions for any entity for which we have derived `gSQL` for its identification (`a`) and entity record (`b`) type.

5.3 Project management example system

In order to test and demonstrate our generic library, we have also implemented the project management system from section 2. This system is a CGI web application written in Clean which runs within an external (Apache) web server and stores its information in a (MySQL) relational database using the GenSQL library. Figure 4 shows the prototype application while updating a project.

5.4 Performance

The generic mapping function relieves the programmer of writing much boilerplate code and SQL queries. It is however important to realize that there is a cost associated with this convenience.

First of all there is some overhead cost in space and time consumption of Clean's generic mechanism. However when optimization techniques [1] are applied by the compiler this can be completely removed.

Secondly there is a cost in the amount of database queries that are performed. The current implementation of the generic operations is not optimized to minimize the amount of queries. Each retrieval or update of an object does a separate query. When an object has many facts with embedded related objects this will result in linearly many queries. Theoretically however, there is no reason why the generic operations would require more queries than handwritten versions.
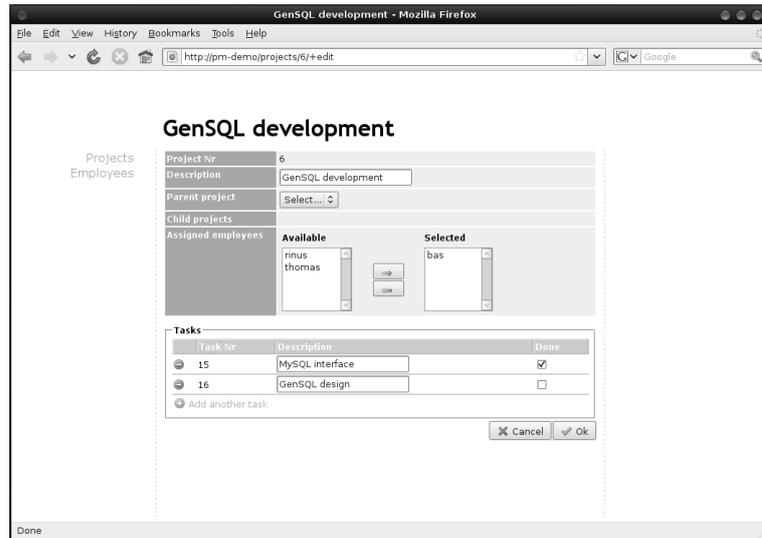
Fig. 4. Screenshot of the project edit page

6 Related Work

At first glance, our library appears very similar to Object Relational Mapping [4] libraries in object oriented languages. These libraries achieve persistence of objects in an OO language by mapping them to a relational database. Although both approaches relieve programmers of the burden of writing boilerplate data conversion code, there is an important difference: our approach treats a subset of all Clean types as a meaningful model of an underlying redundancy free database. This allows us to easily map binary fact types to the entity records of both sides without duplicating any information in the database. In object relational mapping where objects are made persistent, we can only avoid duplication by mapping binary relations between objects to only one side of the relation. Based on this property, object relational mapping is more similar to generic persistence libraries [13] than to the method presented in this paper.

Also related to our work are other methods and tools that use conceptual data models to generate parts of an information system like user interfaces [6], or even complete applications [9]. These tools reduce the effort required to build an information system as well, but are often all-or-nothing solutions that do a certain trick well, but have no solution when you want something a little different. Of course you can always make changes to the generated code, but this means you can only generate once, or have to manually merge your changes upon regeneration. Because our approach is designed as a generic library, and generic programming is an integral part of the Clean language, we can combine a generic solution for common situations together with handwritten code for exceptional situations in one coherent and type safe solution.

The final related area of research is that of abstraction from SQL by embedding a query language inside another language. This approach is used in the HaskellDB library in Haskell [7, 2], in the LINQ library in C# [11], and more recently, using dependent types in a database library for Agda [12]. While these approaches make the programming of data operations easier and type safe, they do not reduce the amount of work one has to do. When using our library, a developer no longer needs to define queries at all, thus eliminating the need for easier and safer ways of defining them. These libraries could however, be used complementary to ours to get a generic solution for the common CRUD operations, and type safety for the exceptional custom queries.

7 Conclusions & Future Work

In this paper we have shown that given the right choice of data types and database tables, it is possible to use generic programming to automate the mapping between entities stored in a database and their representation in Clean.

To do so, we have shifted the focus from both the database and the data types, towards the conceptual level of ORM models. By deriving not only a database, but also a set of types from these models, we enable an automatic mapping between them. This means that by just making an ORM model of a perceived system, you get a database design, a set of types for convenient manipulation, and the machinery for doing CRUD operations on values of those types for free. This relieves a Clean programmer of dealing with how changes in a database must be expressed in SQL, and instead enables the manipulation of a database in a more familiar fashion: manipulation of a local data structure.

We have shown the viability of this approach by means of a prototype library and its use in an example information system. While not ready for production systems yet, this library is already useful for rapid prototyping. But, with optimization of the library, and additional generic operations for handling sets of entities, much of the construction effort of information systems can be reduced to just the definition of ORM models.

What remains to be done is extension of our approach to the complete ORM language. While we selected a subset which is useful for many domains, we have ignored some constructs that make ORM more powerful than, for example, ER. We have yet to investigate how these can be integrated in the current approach.

Another area where further work can be done is to explore how the mechanism for locally manipulating parts of a global shared data structure can be used to facilitate sharing in a functional language. Could it for instance be used to implement a heap on top of an in-memory SQL engine?

Acknowledgements

nomic Affairs. We would like to thank the anonymous reviewers for their constructive comments and suggestions.

References

1. Artem Alimarine and Sjaak Smetsers, *Optimizing Generic Functions*, The 7th International Conference, Mathematics of Program Construction (Dexter Kozen, ed.), LNCS, vol. 3125, Springer Verlag, Jul 2004, pp. 16–31.
2. Björn Bingert and Anders Höckersten, *Student paper: Haskelldb improved*, Proceedings of 2004 ACM SIGPLAN workshop on Haskell, ACM Press, 2004, pp. 108–115.
3. Peter Pin-Shan Chen, *The entity-relationship model—toward a unified view of data*, ACM Trans. Database Syst. **1** (1976), no. 1, 9–36.
4. Mark Fussel, *Foundations of object-relational mapping*, http://www.chimu.com/publications/objectRelational/index.html, 1997, Whitepaper.
5. Terry Halpin, *Information modeling and relational database: from conceptual analysis to logical design*, Morgan Kaufmann Publishers Inc, 2001.
6. Christian Janssen, Anette Weisbecker, and Jürgen Ziegler, *Generating user interfaces from data models and dialogue net specifications*, CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems (New York, NY, USA), ACM, 1993, pp. 418–423.
7. Daan Leijen and Erik Meijer, *Domain specific embedded compilers*, 2nd USENIX Conference on Domain Specific Languages (DSL'99) (Austin, Texas), Oct 1999, Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000), pp. 109–122.
8. Bas Lijnse, *Between types and tables: Generic mapping between relational databases and data structures in clean*, Master's thesis, University of Nijmegen, Jul 2008, Number 590.
9. Elton Manoku, Jan Pieter Zwart, and Guido Bakema, *A fact approach to automatic application development*, Journal of conceptual modeling (2006).
10. Jonathan McCormack, Terry Halpin, and Peter Ritson, *Automated mapping of conceptual schemas to relational schemas*, Proceedings of the Fifth International Conference CAiSE'93 on Advanced Information Systems Engineering, LNCS, vol. 685, Springer Verlag, 1993, pp. 432–448.
11. Erik Meijer, Brian Beckman, and Gavin Bierman, *LINQ: reconciling object, relations and XML in the .NET framework*, SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data (New York, NY, USA), ACM, 2006, pp. 706–706.
12. Ulf Norell, *Dependently typed programming in agda*, Tech. Report ICIS-R08008, Radboud University Nijmegen, 2008.
13. Sjaak Smetsers, Arjen van Weelden, and Rinus Plasmeijer, *Efficient and type-safe generic data storage*, Proceedings of the 1st Workshop on Generative Technologies, WGT '08 (Budapest, Hungary), Electronic Notes in Theoretical Computer Science, Apr 2008.
14. *UML version 2.2 specification*, http://www.omg.org/spec/UML/2.2/, Feb 2009.