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 (ISs) 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 great for efficient storage, it is less suitable for use in the software components that manipulate the data. Therefore, much of the construction of an IS 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, by providing generic versions of the elementary CRUD (Create, Read, Update, Delete) operations. 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. These types reflect the conceptual structure of the entities stored in the database, but also contain enough information about the database to enable automatic mapping. As such, all specific CRUD operations needed in an IS can be automatically derived from the generic definitions and the types by the compiler.
To illustrate the viability of our approach, a prototype library, which implements this mapping, and an example information system have been implemented.

Keywords: Generic programming, Clean, Relational databases, Object role models

1 Introduction

In today's digital society, information systems (ISs) 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. Primarily because, since each organisation has different business processes, information systems need to be tailored or custom made for each individual organisation.

The primary function of information systems is to create, manipulate and view (large) persistent shared data structures. 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. While this representation is great for redundancy free storage, it is less suited for direct manipulation of the data. The reason for this is that conceptually elementary units are often split up into multiple database records. For example, a book consisting of a title and a list of chapters, is broken down into one record for the book and a number of records for the chapters which each reference the book.

When we want to view or manipulate the database on a conceptual level, for example in data entry components, we need data types that represent conceptual units and mapping code to translate between values of that type and the database. As a result, one needs to rewrite many very similar functions for each new IS because each has a unique set of types and tables. This repetitive programming work is not only mind numbing for developers, it is also time consuming and error-prone.

In this paper we present an approach which reduces this work, by providing generic versions of the elementary CRUD (Create, Read, Update, Delete) operations using generic programming in Clean. These operations map changes in data structures that reflect the conceptual structure of entities, to changes in a relational database. A prerequisite for this is that all necessary information about the entities' database representations can be inferred from the types of these data structures. We achieve 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 [5], or a set of types in our approach.

Figure 1 graphically summarizes the mappings in our approach. 1) We derive a set of representation types from a conceptual model. 2) We derive a set of database tables from these types. 3) We use generic CRUD operations for mapping between the two representations. 4) Optionally we can also derive representation types from an existing database instead of a conceptual model.

Our approach 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 ADTs, or on ADTs as interface to a storage representation, we consider ADTs and databases as different representations of the same high-level concepts. Although our approach has a primary application in information systems, it can be applied to a much wider range of systems. Since it maps local changes in a small data structure, to global changes in a larger shared data structure, it can be used to facilitate sharing. Something which is not trivial to realize in functional programming languages.

The remainder of this paper is structured as follows: Section 2 describes how we derive a set of types and a relational database from an ORM model followed by a description of the generic mapping operations between the two representations in section 3. Section 4 illustrates the practical use of our approach through an example, and we wrap up with a discussion of related work in section 5 and concluding remarks in section 6.

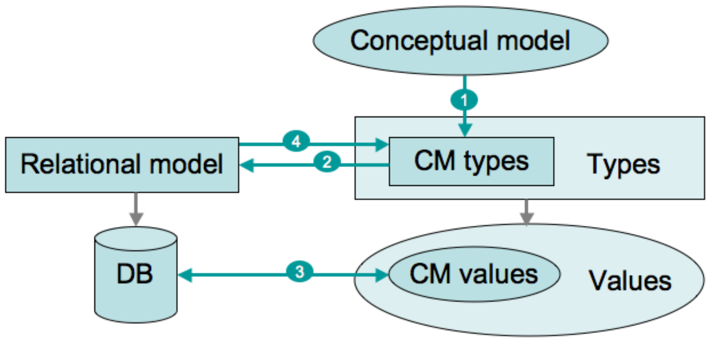


Fig. 1. Deriving data types and database tables from a conceptual model.

2 Types and Tables

The foundation of our approach is the assumption that the data types for which CRUD operations are desired, represent conceptual entities that have some counterpart in a relational database. On top of that, we need to explicitly know the mapping relation between a value of some representation type and its counterpart in the database.

This section presents a method for deriving a set of data types and a corresponding set of relational database tables from an ORM model, such that all necessary information about the tables can be derived from the data types during the execution of CRUD operations.

2.1 Object Role Models

ORM [3] is a conceptual modeling language based on the idea of *objects* playing *roles* in *facts*. With ORM you can model a domain by stating facts about that universe in semi natural language. For example: "**Employee** a *works on* **Project** b". Such facts are then expressed using a formal visual diagram language. This example fact is captured in the ORM model in Fig. 2 which models a small project management system.

For reasons of simplicity, our approach only considers ORM models that satisfy the following constraints:

- The model only contains entity types, value types and fact types. More advanced constructs like subtyping and objectification are not considered.
- The model only contains 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

This subset of the language has roughly the same expressive power as the widely used Entity Relationship (ER) [2] modeling language, with as main difference that in ER identification is not limited to single values.

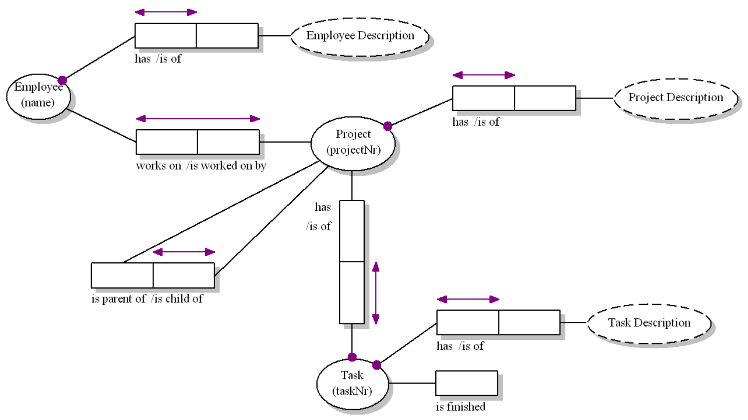


Fig. 2. A simple ORM model for a project management system

2.2 Representation Types

The ORM subset we have defined in the previous section is the input domain of our derivation algorithm. From a programmers perspective however, we are more interested in the output domain of this algorithm: The representation types. Since this is the set of data types that are used in actual programs.

- **Entity Records**
  Records are used as the primary construct to represent conceptual entities. The fields and to use a the name of the entity type it represents. The names of the fields of a record 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 identifier 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 identifier one of the entities are equal to the identity value of another entity. This form is used for one to many relations between entities. The entity identifier 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 identifier is a unique name for this relation and is used by both entity records that have a role in the relation. The match identifiers 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 either be scalar type, record type, `Maybe` of scalar type or record, or list of scalar type or record.
- **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`.
- **Maybe types**
  Clean's `Maybe` type (`::Maybe a = Nothing | Just a`) is used for mapping optional values. These are facts without mandatory role constraints that result in record fields that may not have a value.
- **List Types**
  Lists are used for mapping one-to-many, or many-to-many relationships. It is important to note that the order of these lists is considered to have no meaning in these types. Storage of an entity which contains a list does therefore not guarantee that this list has the same order upon retrieval.

2.3 From ORM To Representation Types

Now that both input and output domains have been defined, we can derive a set of representation types using the following procedure:

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.
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 recursions during execution.

When this procedure is applied to the model in Fig. 2, the following set of representation types is derived:

```
:: 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
                }
```

2.4 From Representation Types to Tables

To get a set of database tables for storage of entities, we could derive a set of tables directly from the ORM model, as is done in the standard Rmap algorithm [5]. However, since the representation types are already close to the relational structure, it is easier to derive the tables from the types. We do this with the following procedure:

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 procedure is applied to the set of representation types in the previous section, we get the set of database tables depicted in Fig. 3.

Note that, because the mapping from types to tables is so direct and preserves almost all information, it is easy to reverse engineer a set of representation types from an existing database. In this case the types can be seen as a view on the database.

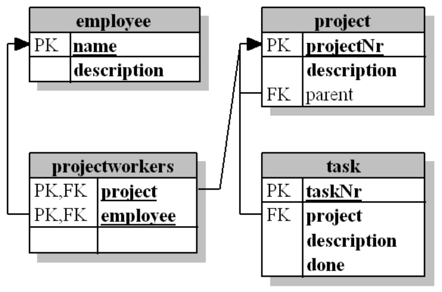


Fig. 3. The derived tables of the ORM model in Fig. 2

3 Generic CRUD Operations

The combination of database tables and representation types defined in the previous section are so directly coupled, that it is now possible to specify the CRUD operations using just the structure of the representation types.

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

```
read   :: id *db     → entity *db
create :: entity *db → id *db
update :: entity *db → *db
delete :: id *db     → *db
```

The type variables `id`, `entity` and `db` are the identification record type, the entity record type and a database cursor type. Since these are variables, a set like this would mean we would have these functions available for every entity and identification type and for every type of database.

This section first describes on a high level how these four operations can be defined on the structure of the representation types. It then proceeds with an account of how we have implemented them in a prototype library in Clean using generic functions.

We cover the operations in the order "read, create, update, delete" instead of "create, read, update, delete", because the read operation illustrates the basic principles behind these operations best, and is therefore explained first.

3.1 The Read Operation

The read operation reads all information about one single entity from the database. It is given a value of an identification record type and constructs a value of the corresponding entity record type. This is done using a process very similar to recursive descent parsing of a string or file. In both cases a nested data structure is constructed from a flat sequence of tokens. A big difference however, is that instead of having the input stream completely available when we start parsing, we can only construct the input stream just in time during parsing. This is because the input values are not stored together as a sequential stream, but are scattered over different tables in the database and we need information about the type of the value we are constructing to locate it.

As we have seen in section 2, the building blocks of our representation types are scalar values, records, lists and `Maybes`. We can now see how each of these is constructed from a list of tokens.

- **Scalar Values**
  The construction of scalar values is the simplest part of the read operation. It consists of nothing more than checking if the value is available at the head of the token list and removing it.

- **Records**
  While the construction of scalar values is necessary to create the "leaves" of our data structures, they are of little use without construction of the "nodes" that define the structure. The records in the representation types are the building blocks which provide this structure.
  The construction of records consists of recursively constructing all of its fields. But the values required to do so are not in the token stream yet. The only value that is in the token stream is the value of the first identifying field. The construction is therefore split up into two steps:
  1. Fetch data for all fields from the database. Using the information encoded in the record fields' names and their types we perform a number of SQL queries to retrieve the values for all fields and add them to the token stream. When a field's type is a list, a special terminator token is also added to the stream to indicate the end of the values for that field.
  2. Construct the record. With all data in the token stream we can now recursively perform the create operation for all record fields.
- **List values**
  Construction of lists is done by recursively constructing elements until the head of the token stream is the terminator token. At this point an empty list is constructed.
- **Maybe values**
  When we need to construct a `Maybe` value we just check if the head of the token stream is a `NULL` value. If it is we return `Nothing` and are done. If not, we recursively construct a `Just` value.

With the above procedure we are able to construct an entity record from a token stream and a database. The only remaining step is the initialization of the token stream. This is achieved by flattening an identification record to a singleton list of tokens using a procedure very similar to the "create" operation which we cover next.

3.2 The Create Operation

The "create" operation creates the database records for a data structure which has no counterpart in the database yet. This is achieved by doing more or less the inverse of the "read" operation. Where the read operation can be viewed as a parsing problem, the create operation can be viewed as a printing problem. In this case we "print" a data structure to a stream of tokens.

There is a complication however, that makes the "create" operation somewhat more complex than the "read" operation. The order in which records are read from the database does not matter, but the order in which they are created does. The reason for this is twofold: We want to keep the ability of using *auto incrementing* fields in our database, and we are only allowed to refer to existing database records because of foreign key constraints. When an entity is identified by an auto incrementing field in the database, we can only refer to it after its primary record has been created. This is because we simply do not know what identification value that entity will have until it is created in the database. Even when we know what the identification value will be in the database we are still not allowed to use it in fields that have a foreign key constraint when the referred record is not created first. Therefore, the order in which the various parts of an entity record in Clean are mapped to the database is important during this operation.

Now we can look at how each of the building blocks are mapped to the database during the create operation:

- **Scalar Values**
  For scalar values we do the exact opposite of the "read" operation. Instead of reading tokens and constructing values, we deconstruct values into tokens and add them such that they can be stored in the database when processing records.
- **Records**
  Because of the ordering issues with mapping records to the database the recursive "create" operations for the record's fields is split up in two passes. In between those passes the creation of records in the database is performed. Hence, the operation consists of the following three steps:
  1. Do the first recursive pass. During this pass all fields that are contain scalar values or entities that the current entity will point to will be created. After this pass all the necessary tokens have been added to the stream such that a new record can be created in the database.
  2. Map the current entity to the database. The tokens from the first pass are used to create a record for the current entity in the database. We now know what the identification value for this entity is.
  3. Do the second recursive pass. During this pass all fields which contain entities that refer to the current one are created. They can either refer directly from the records that represent the entities in the case of one-to-many relationships or be linked to the current entity through an additional table. In the latter case the records in the linking table are also created during this pass.
- **List Values**
  The "create" operation for lists is just the repeated "create" operation for each of the elements in the list followed by an addition of the terminator token to the token stream.
- **Maybe Values**
  Maybe values are also simple again. A `Nothing` adds a `NULL` token to the stream and a `Just` recurses to yield the tokens.

When the above procedure is applied to an entity record, we map the whole data structure to a representation in the database and end up with a single value that identifies the entity in the token stream. This single value is then converted to an identification record by applying the "read" operation to the token stream.

3.3 The Update Operation

The update operation is similar to the create operation in the sense that it also traverses a data structure and writes the values to the database. There is a difference however, that makes it the most complex of all four operations. When a relation between entities is mapped to a field in which the related entities are included as a list of records, we need to deal with the addition of new entities or removal of existing entities in that list.

As with the read and create operations, we cover the update operation for each of the building blocks of the representation types separately:

- **Scalar Values**
  The update operation for scalar values is equal to that of the create operation. It merely converts scalar values into tokens.
- **Records**
  Because of the additional complexity of dealing with lists of embedded entity records, the update operation is performed in three passes. The first and second pass are practically identical to the "create" operation, but before the first pass, the fields of the record are read from the database and kept until after the second pass. They are then used to detect if new entries have been added or existing ones have been removed from list fields. We can then conclude with a third pass in which the "create" operation is applied to the new items and the "delete" operation on the removed items.
- **List and Maybe Values**
  Like the scalar values, updating list and maybe values is similar to creating lists. For lists, the update operation is applied to each member of the list after which a terminator token is added. Maybe values add a `NULL` token to the stream when `Nothing` or recurse when `Just`.

3.4 The Delete Operation

The final operation which completes our set of basic operations, is the delete operation. This operation is very similar to the read operation. Since it needs to locate all data in the database about a given entity in order to delete it. It is just as easy to yield this data as result of the operation.

When deleting entities from the database, we have a reversed version of the order limitations of the "create" operation. We are only allowed to remove records from the database when they are no longer referenced by other records. We therefore apply a multi-pass strategy again.

For the last time, let's look at the different building blocks again.

- **Scalar Values**
  Deleting scalar values is the same as reading scalar values. Tokens are taken from the stream and scalar values are constructed.
- **Records**
  Records are deleted using a two pass strategy. During the first pass all fields which do not have values linking back to the current entity are deleted and

read. Fields that can not be deleted yet because of the required ordering are only read. After this pass the record representing this entity in the database is removed. We can now finish with a garbage collect pass which removes everything that could not be removed in the first pass.

– **List and Maybe Values**
  Deleting list and maybe values simply follows the same procedure as the "read" operation on lists and maybes.

3.5 Implementation in Clean

To validate the generic operations, we have implemented the operations described in the previous as a prototype library in Clean. This library called "GenSQL" implements these operations as described, without any optimizations for better performance.

Jack of All Trades. Because the generics mechanism in Clean has some limitations, the actual implementation of the operations in the GenSQL library has a somewhat unusual design. In Clean it is not possible for generic functions to call other generic functions. 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 modes are either one of the four operations `GSQLRead`, `GSQLCreate`, `GSQLUpdate`, `GSQLDelete` or the type information mode `GSQLInfo` or `GSQLInit`. This last mode is an extra operation which "prints" a reference value to the token list in order to start a read or delete operation.

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`. This is the generic type variable, which means that it is different for each type of value we apply this function on. 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 "printed" to the token or info list. The second argument is the token list. In this list tokens are accumulated during the "printing" or "parsing" of the data structures. The third argument is the info list. In this list, the type information about record fields is accumulated.

The last argument of the `gSQL` function is a unique database cursor. This is a handle which is used to perform queries and statements on the database.

The return type of the `gSQL` function is a tuple which contains an optional error and the possibly modified value of type `t`, the token list, the info list and the database cursor.

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 self explanatory wrappers are defined as follows:

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

4 Example

To illustrate how the generic CRUD operations of section 3, and especially their prototype implementation, can be applied in the construction of an information system, we have implemented a small project management system that uses them.

This system 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.

To apply our methods, one needs an ORM model of the intended system. The model describing this example is given in Fig. 2. Next the data types have to be derived using the procedure of section 2. The types we get for this example are shown in section 2.3. Finally we need to derive the corresponding tables, which are shown in Fig. 3. We now have everything in place to use a **derive** `gSQL` on our types and let the compiler generate the CRUD functions.

Though this example system is fairly small, it does cover all of the constructs we defined in our ORM subset, and therefore serves as a useful testcase.

4.1 Application Design

The basic design of the example system is that of a CGI web application which runs within an external (Apache) web server and stores its information in a (MySQL) database. Pages in the application are modelled as functions, where each page is a function which generates a page from an HTTP request and a database connection.

The libraries this system uses are: An HTTP library which manages the CGI interfacing, an HTML library for the construction of HTML documents, a MySQL interface and of course the GenSQL library for the CRUD operations. Everything else is custom code.

4.2 The Operations in Action

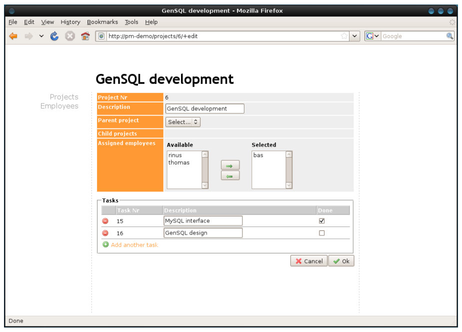
The best way to illustrate the use of the GenSQL library, is by seeing it in action. Figure 4 shows the edit page for a project, which is implemented by the following function:

```
editProjectPage :: !Int !HTTPRequest !*cur
    → (Maybe (String,String), !String, [HtmlTag], !*cur)
    | SQLCursor cur & bimap{|*|} cur
editProjectPage pid req cursor
    | req.req_method == "POST"
        ♯ project                       = editProjectUpd req.arg_post
        ♯ (mbErr,mbId, cursor)          = gsql_update project cursor
        | isJust mbErr
            = (Nothing, "Error",[Text (toString (fromJust mbErr))],cursor)
        = (Just ("/projects/" +++ toString (int (fromJust mbId)),
             "Successfully updated project " +++ toString pid),"",[],cursor)
    | otherwise
        ♯ (mbErr, mbProject, cursor)    = gsql_read pid cursor
        | isJust mbErr
            = (Nothing, "Error",[Text (toString (fromJust mbErr))],cursor)
        | isNothing mbProject
            = (Nothing, "Error",[Text ("There is no project with project nr "
                +++ toString pid)], cursor)
        ♯ project                       = fromJust mbProject
        ♯ (projects, cursor)            = getProjectOptions cursor
        ♯ (employees,cursor)            = getEmployeeOptions cursor
```

```
      = (Nothing, project.project_description,
          [editProjectForm False project projects employees],cursor)
```

The structure of this function is relatively straightforward. When a form is posted, the data in that post is parsed to create a `Project` data structure. Then, the generic mapping is used to propagate the update to the database, and the user is redirected back to the "show" page with a friendly message. When nothing is posted, the generic mapping is used to read a project from the database, and a form is created.

Most of the code in the above function deals with showing messages and other trivialities. The only interesting parts of are the calls to `editProjectUpd`, `editProjectForm`, `gsql_read` and `gsql_update`. The first two functions are not trivial, but have an ad-hoc implementation for our types. The last two functions are the wrappers of the GenSQL library that implement the generic database operations. Because the nested structure of the `Project` type is not visible in this function, the complexity of these operations is not immediately apparent. However, the `Project` data structure is constructed using information from three different tables and requires updates in all of those tables and some garbage collection when a project is edited.


GenSQL development - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://pm-demo/projects/6/+edit
Google
Projects
Employees
Cancel
Ok
Done

GenSQL development

Project Nr	6
Description	GenSQL development
Parent project	Select...
Child projects	
Assigned employees	Available: rinus, thomas; Selected: bas

Tasks

Task Nr	Description	Done
15	MySQL interface	✓
16	GenSQL design	☐

Add another task

Fig. 4. Screenshot of the project edit page

5 Related Work

There are two areas of research that are related to the approach presented in this paper. Although we are not aware of any prior work that uses the generic programming method of functional languages like Clean and Haskell to do an automatic mapping between nested data structures and relational databases, it is similar to "Object Relational Mapping" known in object oriented languages. This technique tries to achieve persistency of objects in an OO language by mapping them to a relational database. Our approach differs from this technique in that we consider data structures as temporary representations of the conceptual entities in the database, instead of considering the database as a means of storing data structures.

Another slightly related topic is the research on creating a kind of type safe SQL in several functional languages by embedding it in the language as a DSEL. This approach is used in the HaskellDB library in Haskell [4, 1] and more recently, in the dependently typed language Agda [6].

6 Conclusions

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.

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 could be reduced to just the definition of ORM models.

Acknowledgements

This work has been partially funded by the Technology Foundation STW through its project 07729: "Demand Driven Workflow Systems".

References

1. Björn Bingert and Anders Höckersten, *Student paper: Haskelldb improved*, Proceedings of 2004 ACM SIGPLAN workshop on Haskell, ACM Press, 2004, pp. 108–115.
2. Peter Pin-Shan Chen, *The entity-relationship model—toward a unified view of data*, ACM Trans. Database Syst. **1** (1976), no. 1, 9–36.
3. Terry Halpin, *Information modeling and relational database: from conceptual analysis to logical design*, Morgan Kaufmann Publishers Inc, 2001.
4. 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.
5. 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.
6. Ulf Norell, *Dependently typed programming in agda*, Tech. Report ICIS-R08008, Radboud University Nijmegen, 2008.