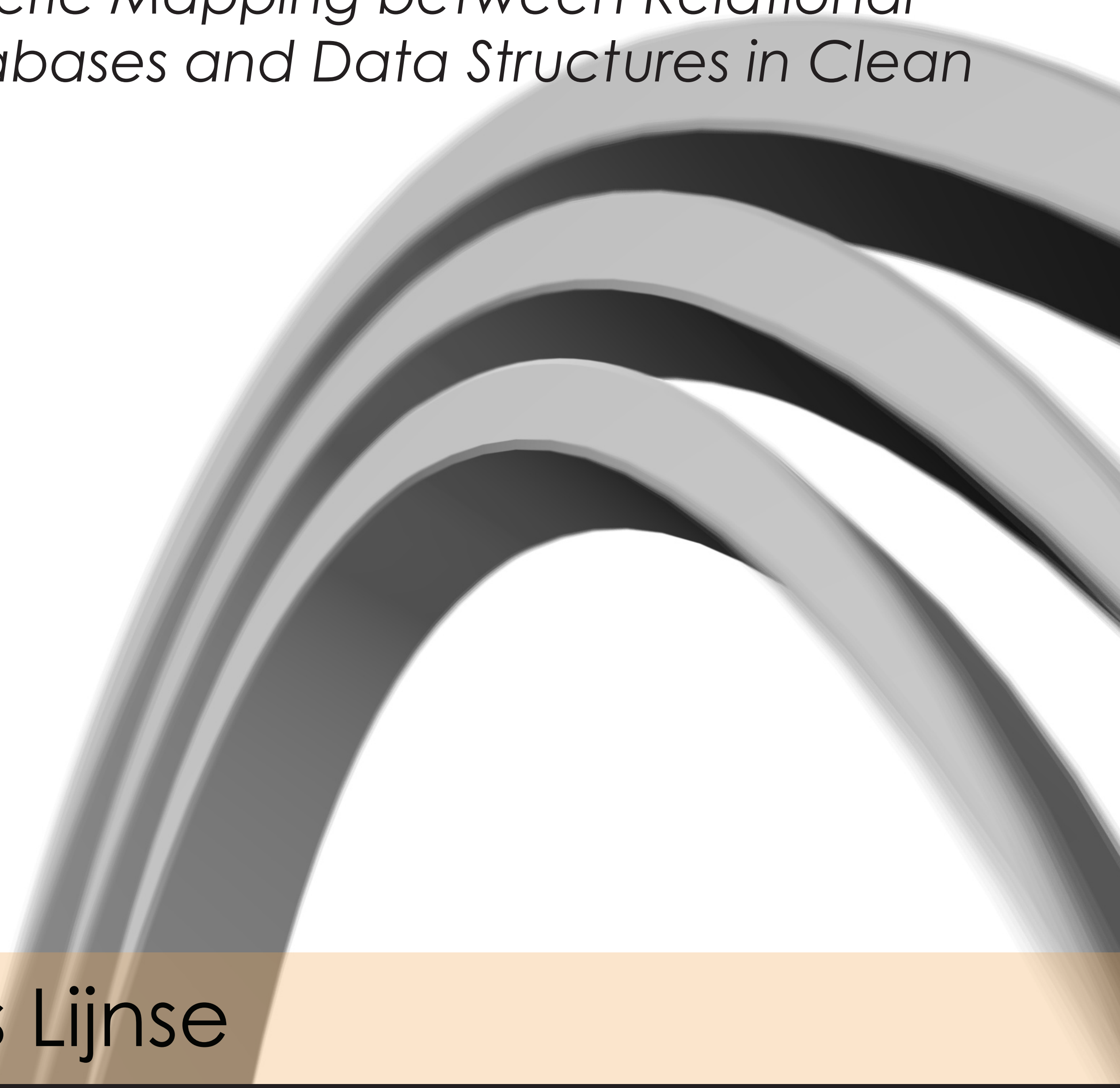
Between Types and Tables

Generic Mapping between Relational Databases and Data Structures in Clean



Bas Lijnse

Master's Thesis Computer Science
Radboud University Nijmegen
Thesis Nr 590, July 2008

Supervisors
prof.dr.ir. M.J. Plasmeijer
dr. S.J.B.A. Hoppenbrouwers (reader)

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. Since each new information system requires the same kind of operations, but for different types of data, much of this effort consists of repetitive programming work.

In this thesis we explore how generic programming in Clean can be used to reduce this effort. The presented approach uses Object Role Models to systematically derive both the relational model of a database, and the types of the data structures that represent entities in that database. In doing so, a clear relation between these types and the database is maintained, which enables automated mapping between them.

To support this approach, a prototype library, which implements this mapping, and an example information system have been implemented.

Keywords
Generic programming, relational database, object role model, Clean

Contents

Chapter 1

Introduction

In today's digital society, information systems (ISs) play an important role in many organizations. A lot of administrative business processes are supported by such systems, and some 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 bulk of work in the IS development process can be divided into two groups of activities: specification and design, and software construction. During the specification and design activities the knowledge embedded in business processes is made explicit and codified in written specifications. These specifications consist of natural language requirements and sometimes, more or less formal models of the conceived system. Because computers are not able to observe business processes or interview domain experts, these activities will always require human effort. Specifications are usually written in natural language and cannot be interpreted directly by computers. Therefore, the software construction activities are needed, during which the specifications are interpreted by programmers who translate them into computer programs. The amount of work required to do so largely depends on the level of abstraction that the used tools and programming languages offer. The higher the abstraction, the more concise the specifications can be expressed.

A technique which can be used to reduce programming effort in systems where similar operations are defined for many different data types is "generic programming". This technique allows the specification of high level algorithms that work for *any* type. Since IS development is a largely standardized and well understood process, a lot of the construction work involved is repetitive. For example, there is little difference between the construction of a data entry interface for the entry of new patients in a hospital IS and the entry of books in a library IS. In both cases, the steps in a data entry interface are the same: Read data from a number of database tables, construct some data structure, manipulate that data structure (e.g. by presenting it to a user as a form), and finally propagate the changes back by updating, adding or deleting records in a number of database tables. The data structures used during manipulation are essentially representations, or views, on parts of the database. Due to the difference in domains, the representation data types and database tables for patients are different then the ones used in the library IS, which means we have to write separate code for both systems. An interesting question is now: How can generic programming be used to reduce the construction effort of these data entry parts of an information system?

A different approach to reducing construction effort, is reuse of the specification effort. If specification is done using formal languages and models instead of just natural languages, automated transformation of specifications into parts of the executable system becomes possible. Formalization of the specification process also introduces other advantages such as the possibility of automated checking of properties such as ambiguity or contradiction. A formal modeling language that is useful for IS design 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 is possible to derive a relational database schema directly from the model. While this already an example of reuse of the specification effort, one might wonder if ORM models could be used to derive even more parts of an information system.

At first glance, these two approaches are very different. But if we look closer, we see that there is a connection. If we want to use generic programming to automate the mapping of data from the various tables in the database to the representation data structures, we need to know the relation between the types and tables. For arbitrary types and tables this relation is not clear, or does not even exist. But when we know that both the types and the tables represent the same concepts we might succeed. This is where we can benefit from ORM models. If we use an ORM model, to not just derive a database, but at the same time derive the data types that will serve as representations for data entry, we are able to maintain an explicit relation between the database and those types. In this thesis we explore this approach by answering the following question:

> How can we derive a database *and* a set of representation/view types from an ORM model, such that generic programming can be used to automatically map between them?

1.1 Background

The foundations on which this thesis builds, are on the one hand conceptual modeling, especially Object Role Modeling, and on the other hand generic programming. Before we continue with the further definition of our research question in the next section, we briefly discuss both of these techniques to provide the required context.

1.1.1 Object role modeling

In order to find a mapping between entities that have some storage representation in a database and another representation defined by their types in a programming language, we have to "zoom out" to the conceptual level to see what the entities and their properties and relations actually are, because both representations are simply different forms of the same conceptual entity. ORM [8] gives us a formal modeling language in which we can describe the "world", or so called Universe of Discourse (UoD), of our information systems at this conceptual level.

ORM is a conceptual modeling language based on the idea of *objects* playing *roles* in *facts*. It is a formalization of its predecessor, NIAM (Natural language Information Analysis Methodology) [20], which is a similar conceptual modeling technique. With ORM you can model a UoD by stating facts about that universe in semi natural language. For example: "**Person** a *works for* **Department** b". Such facts are expressed using a formal visual diagram language. It is also possible to "verbalize" facts in a model. This means that the semi natural language sentences describing them can be derived from the graphical model automatically. This is a very useful feature for the validation of models by domain experts.

Unlike other conceptual modeling techniques, such as UML class diagrams [1], ORM has a well defined syntax and semantics. ORM models are therefore not only well defined specifications, but also have *formal meaning* and can be used as input for the derivation of other models. The standard example of this is the derivation of relational models from ORM with the Rmap algorithm [13], but we can also use it to derive other models, such as classes in an object oriented language, or in our case types in a functional language.

1.1.2 Generic Programming

Generic programming is a term which has many meanings in many different contexts. The shared idea behind all of them is that one solution can be specified that is applicable to many similar problems. The actual techniques that are used to achieve this goal however, vary wildly. For example the generics introduced in Java 1.5 [5] are just parametric polymorphism. Something which has been available in functional languages for a long time.

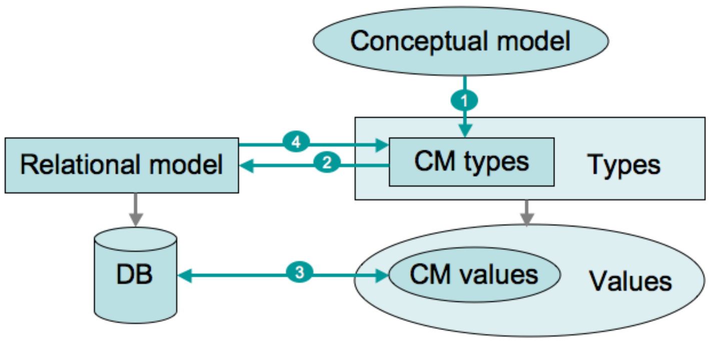


Figure 1.1: The five sub problems to be solved

Even within the family of functional languages, the term generic programming is used for different techniques. The shared context here, is that it is used for techniques that allow a programmer to specify functions that can be applied to values of *any* type. In this thesis we will use the generic programming mechanism of Clean [2]. While similar mechanism exist in, for example Haskell [10], Clean has the advantage of having generics built into the language and the compiler directly. This allows us to write and use generic functions without depending on additional preprocessors or compiler extensions.

Generic programming in Clean is based on the idea that any value can be systematically transformed to a generic domain and back. This generic domain consists of a three types which can describe any value: A *unit* which is a non parametrized value, an *either* which represents a choice, and a *pair* which enables composition of values. Once a programmer specifies a function which is defined for values of the types of the generic domain, he has a function which is applicable to values of any type. A value is first transformed to the generic domain, then the generic function is applied and finally the modified generic value is transformed back to it's original type. In Clean, the transformation from and to the generic domain is handled transparently by the compiler. The only thing we have to do is write generic functions and tell the compiler to *derive* that function for the types we want to apply it to.

1.2 Problem definition

Since our question is too broad to tackle at once, we apply a "divide and conquer" approach. We realize the complete generic database mapping by dividing the problem into sub mappings. These are depicted in figure 1.1, in which the arrows indicate the sub mappings. The numbers on the arrows correspond to sections 1.2.1 to 1.2.4 in which each is explained further.

1.2.1 Mapping conceptual models to types

The first problem that needs to be addressed is the question of how to represent entities in the in the Clean programming language. Because information in an IS is updated incrementally we need to find a representation for single entities, instead of the entire database.

The challenge in this problem lies mostly in designing the representation in such a way that it can be readily used by the programmer in a convenient way. If this is not the case, it still is necessary to manually program transformations between the representation used by the mapping, and convenient custom data types. In this case little is gained, because instead of translating between custom data types and the database language (SQL), the programmer now has to translate between his own data types and the representation used by the automatic mapping.

Because we use generic algorithms for the manipulation of the representation, we can use a representation scheme in which each model is represented by its own set of types. This freedom in the representation scheme improves the usability of the representations because it is not necessary to represent every possible model using a fixed set of types.

Since this mapping takes us down from a conceptual to an operational level, the mapping may involve implementation choices by the IS developer. The aim for this sub mapping is therefore not to provide a fully automated mapping, but just a description of a systematic approach that may be applied to derive useful types from an ORM model.

1.2.2 Mapping types to relational models

Once we have a way of representing entities in an IS as convenient data types, we have to design a mapping from these types to a suitable relational database representation. Because the representation types contain all information about the database, we can derive the relational structure of the database from that embedded information.

Since the relational structure can be derived purely from the types, we can, in theory, automate the derivation process completely. However that requires the representation types to be first order values. Because they are not, and this mapping needs to be done only once for each new system, we again aim only at a description of the derivation algorithm.

1.2.3 Mapping operations to SQL

To perform the basic operations on a database, create, read, update, and delete (CRUD), we need to be able to fetch all data required for that operation from the database. Then we are able to create instances of the representation types. We also need to be able to translate our representations back to SQL statements, to actually perform the changes in the database.

The generation of this SQL code is a non-trivial exercise because a single update in a moderately complex representation type, can require the execution of a series of `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements in exactly the right order. Since the required information about the database is embedded in the types, we need some advanced generic functions to perform these operations.

Because of the complexity of the generic functions required to perform the operations, we do not just give a description of how the operations can be performed, but implement them in a prototype library as well. This prototype library then serves as a proof of concept of the proposed mapping.

1.2.4 Mapping relational models to types

For the design of new information systems and databases, we can make an ORM model and use the previously proposed mappings. But not all existing databases are designed using an ORM model. If we want to use the mappings for existing databases, we need a different approach.

When no ORM model is available we have to derive the representation types from a different source. The obvious choice is then the relational model, because it is always implicitly available and can be extracted from a relational database. Such models do however, contain less information than a conceptual model because the concepts from the UoD have already been flattened into a tabular structure. Nonetheless, we explore if we can make a mapping from these less detailed relational models, to the representation types.

Again, we are only interested in a description of the mapping.

1.2.5 Testing by example

The best way to find out if a programming method is possible, is by just trying it out. In this project we test our complete mapping approach by using it in an example system. The specifications of this example system are given in section 1.4.

1.3 Related work

There are two areas of research that are related to the approach presented in this thesis. While no prior work exists which uses the generic programming method of functional languages like Clean and Haskell to do an automatic translation between nested data structures and relational databases, it is similar to a technique called "Object Relational Mapping" known in object oriented languages. Another slightly related topic is the research on type safe SQL in several functional languages.

1.3.1 Object-relational mapping

Object-relational mapping [7] is a technique in object oriented languages which tries to automatically map objects to records in a relational database. While this technique is commonly used in industry, it is very little documented in scientific literature. This may be due to the fact that it is a largely pragmatic approach without a solid theoretical foundation.

The idea behind object relational mapping is to use a relational database as a persistent store for objects by mapping classes to database tables. A piece of software called an object-relational mapper then provides a store and load operation for objects which maps them to a database. Such a mapper often provides caching facilities which keep objects in memory. So when an object is loaded twice, the *exact* same object is returned by the mapper. This is important because in a relational database, relations are considered equal when all values in the record are equal, but in an object oriented language, each object has its own identity even when all attributes are the same as another object. This discrepancy between the behaviour of objects and relations is known as the "object-relational impedance mismatch" and it is sometimes argued that correctly mapping classes to relations (tables) is impossible because of this mismatch. Another problem of object relational mapping is the issue of dealing with inheritance. If an object `x` is of class `B` which is a sub class of another class `A`, the object is both of class `A` and of class `B`. When both classes define attributes, it is not clear how the object `x` should be mapped. Different object relational mappers, therefore use different strategies and heuristics to determine a relational schema in which `x` could be stored persistently.

While the approach described in this thesis is similar to object relational mapping, there are some important differences.

- The objective is not to use a database as a persistent store for data structures, but to use data structures as a temporary representation for manipulating a database.

- Types in a functional language are different from classes in an object oriented language. A mapping between types and relations is therefore different in nature than a mapping between classes and relations.

1.3.2 Type safe SQL

A very different approach to improve the way of programming with relational databases in functional language, is the modeling of SQL queries with the type system of the language. This allows the type checker to find errors in an SQL query and thus introduces compile time verification of database queries. This approach has been used in the HaskellDB library in Haskell [11, 4] and more recently, in the dependently typed language Agda [15].

These approaches are related because they also eliminate the need to manually code SQL statements, but are limited to the relational domain. Using such systems, the programmer still has to define all the queries and data transformations needed to store a complex nested data structure in a collection of flat tables.

1.4 Example: A Project Management System

To illustrate the mapping procedures in the upcoming chapters, we use a running example. This example is a simple project management system. To provide the necessary context, we briefly introduce its specification here. In chapter 6 we show an implementation of this system which uses the mapping.

In the project management system we have 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 it's own.
- **Tasks** are units of work that have to be done for a certain project. They are identified by a unique task number and also have a textual description. The system should also keep track of whether a task is finished or not.
- **Employees** are workers that are identified by a unique name and also 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.

An ORM model of this system is given in figure 1.2.

While this example system is fairly simple and does not use the more advanced constructs of ORM like subtyping, objectification or n-ary fact types, it does have some features which make it an interesting enough example.

- The system has facts about entities and values, as well as facts about relations between entities.

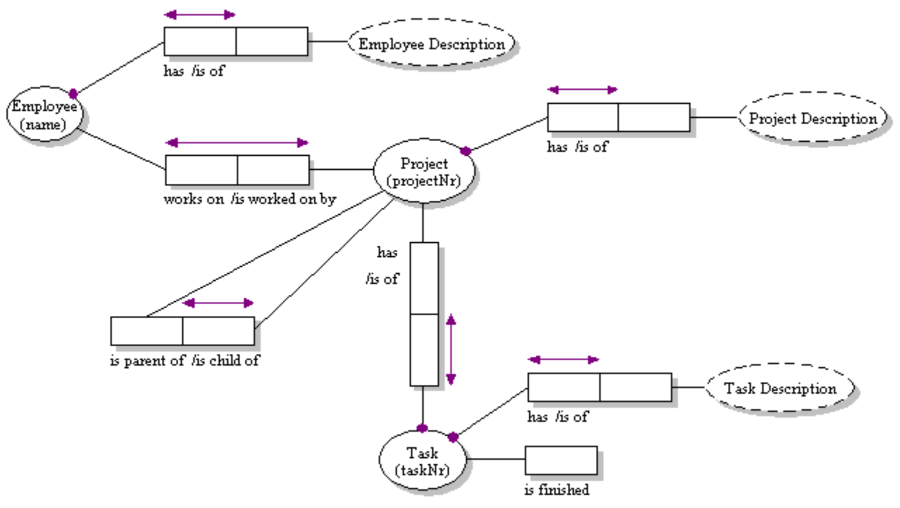


Figure 1.2: A simple model for a project management system

- Beside binary fact types, the model also has a unary fact type.
- The system contains constraints which allow many-to-one and many-to-many relations
- The system contains a binary fact about an entity type with itself

It is trivial to add more "attribute" like facts to the model, or add more relations between entities, but this only makes the model larger, and does not introduce new constructs to deal with in the mapping. Allowing constructs such as subtyping or n-ary roles would be interesting to have in the example but have been left out for the sake of simplicity.

Chapter 2

Mapping conceptual models to types

2.1 Introduction

At first glance, finding a representation for conceptual models seems a trivial task. One could simply map the conceptual model to a relational model, which would be required for storage in a relational database anyway, and then represent each table in the relational model as a record in Clean. The obvious drawback of such an approach is that the representation reflects the relational structure, rather than the conceptual structure of the domain at hand. To gather all information about a certain concept, requires the programmer to manually retrieve information from different tables to create a data structure that reflects this concept. When the goal of the representation and the associated mappings is to relieve the programmer of the burden of translating between the world of Clean types and the world of relational databases, such a representation is of little help.

So, not all representations are equally suitable. But what makes a good representation? The answer to that question depends on the intention of the programmer, because the goal he is trying to achieve defines the requirements for the data types he uses. To find one single representation scheme requires generalization over these goals. If we assume that the goal of the programmer is to manipulate the populations of some conceptual model that are stored in a relational database, we can define a set of three basic operations that the representation must support:

- Easily create instances of conceptual object types.
- Easily update instances of conceptual object types.
- Easily delete instances of conceptual object types.

These operations focus on entity types of a conceptual model because they reflect the concepts in a universe of discourse. They abstract from the tabular structure used for storage and represent the meaningful entities one talks about. For example, in a model with the concepts **Person** and **Company** which each have a name, one is not so much interested in names alone, but only in the context of the concept **Person** or **Company**. So, hopefully the choice of using conceptual entity types as the primary unit that has to be represented in Clean, will give us a representation and mappings which will make the programming involved in building information systems easier.

Other aspects that have to be taken into account in the design of the representation scheme, are the possibilities and limitations of Clean itself. For example, Clean does not have a standard way of using references, so these have to be explicitly taken into account in the representation.

Due to the rather vague goal of "making it easier" for a programmer to manipulate the population of an information system, the design of the representation and mapping scheme, as many programming problems, cannot be derived straightforward from the goals but requires some programmer's intuition and common sense. We therefore approach this problem by first defining a simple solution for the basic cases and then evaluate it against the sub goals given earlier. Based on this evaluation, the representation and mapping are extended to the more general case.

2.2 A first approach

To start the search for a good representation, we begin with a limited set of models and see if we can find a suitable representation for them.

The minimal set of models to start with should at least contain entity and value types. These are the atomic entities on which the rest of the model is built. Obviously, we also need to have fact types in the set. We limit the fact types to just unary and binary types, to keep things simple. The last thing we need to make a sensible first approach, are uniqueness and total role constraints. Without these we have no knowledge at all about how many times objects may or should participate in certain facts. To decide how a concept should be represented, we should at least know if certain values always have a value, or if there is one, or many values.

For these simple models without complex constraints or fact types with more than two roles, we derive a set of types using the following straightforward algorithm.

1. Define a Clean record type for all object types in the ORM model.
2. In each of these records add fields for the identification of that object type and all facts in which it participates.

3. Determine the type for each of the fields based on the fact type and constraints on the fact type. Label types are represented by actual Clean types, while object types are represented by special identification types.

   - Unary fact types are represented as `Bool`
   - Binary fact types without a uniqueness constraint on the role of this entity type are represented as a list of the identification/value type of the other role.
   - Binary fact types with a uniqueness and a total role constraint on the role of this entity type are represented by the identification/value type of the other role.
   - Binary fact types with a uniqueness constraint but without a total role constraint on the role of this entity type are represented as maybe the identification/value type of the other role.

4. Define an identification type for all entity types in the model. This is achieved by creating a type synonym for the type of the identification of the entity type. This is either a scalar type, or a tuple of scalar types when an entity type cannot be identified by a single value.

This algorithm is best illustrated with an example. If we apply this algorithm to the model in figure 1.2 we derive the following set of Clean types:

```
:: Employee =   {   name            ::  String
                ,   description     ::  String
                ,   works_on        ::  [ProjectID]
                }
:: EmployeeID   :== String

:: Project  =   {   projectNr       ::  Int
                ,   description     ::  String
                ,   parent          ::  Maybe ProjectID
                ,   children        ::  [ProjectID]
                ,   tasks           ::  [TaskID]
                ,   worked_on_by    ::  [EmployeeID]
                }
:: ProjectID    :== String

:: Task     =   {   taskNr          ::  Int
                ,   description     ::  String
                ,   is_finished     ::  Bool
                ,   project         ::  Int
                }
:: TaskID       :== Int
```

2.3 Problems of the first approach

While the basic algorithm of the previous chapter already provides a reasonable representation of the concepts in the model, we can see, even in this simple example, that this approach is not good enough yet. Suppose we wanted to always have the full list of tasks as part of the project record instead of references to tasks. We would than prefer the type of the `tasks` field in the representation of projects to be `[Task]` instead of `[TaskID]`.

This example shows that a fully automated approach is not desirable for the mapping of conceptual models to Clean types. A designer should be able to guide the mapping making design decisions based on the intended use of the types.

We will now look at several properties of our first approach that are problematic in real use.

2.3.1 The conceptual model subset

In our first approach we have deliberately limited our mapping algorithm to a subset of ORM. While it is of course desirable to provide a mapping for the entire ORM language, it is simply too much work to deal with all the details. This additional work merely distracts from the main research objectives.

We do however, need to be more clear about exactly what the subset of ORM is that is covered by the mapping. Our first approach did not describe all possible cases that are possible with the ORM constructs mentioned that were said to be allowed.

Our final mapping may be defined for a subset of ORM only. But this subset should be unambiguously defined and completely covered by the mapping.

2.3.2 Design choices

As became clear in the first example of this chapter, the suitability of the representation depends on the intentions of the programmer. The mapping should therefore allow the programmer to make important design choices during the mapping process.

Reference or inclusion

An important choice that has to be made by the programmer is what type is used for fields in the records representing a fact with an entity type in it. One could choose to simply reference entity types, but another option is to use the actual type representing that entity.

Whether a reference to another object, or an inclusion of that object in the record is chosen depends largely on the nature of the concepts that are represented. If an object type has a small representation type, for example a user account which consists of a username and password, it makes sense to include it directly into another type. But when an object type is represented by a large type or when many instances of a fact may be linked to an object, it sometimes makes more sense to use references.

Thus, depending on the circumstances the mapping must allow the programmer to choose between references and inclusions in the definition of records representing object types.

Choosing to use inclusion of other entities in an entity's record is not without hazards. One must be careful not to introduce inclusion cycles. If an entity type indirectly includes its own type, operations on that type will no longer terminate. This happens because, for example, reading an entity of type `A` from the database which includes another type `B` causes all related entities of type `B` to be read as well. If `B` again includes `A` this causes all related entities of type `A` to be read as well. This in turn causes all type `A` entities to be read and so on. Similar problems occur with the other operations.

The mapping must therefore only allow the designer to choose inclusion instead of reference when this does not introduce inclusion cycles.

Fact subsets

For large conceptual models, the representation types derived using the simple approach could turn into quite large records. This would also mean that to make an instance of such a type, all that information has to be retrieved from the database. Depending on the facts, this could be a relatively expensive database operation. But it may be that for a certain program, only a few of the fields of the record are used. This would make retrieving and storing of the entire records a waste of resources.

So for efficiency reasons and just making the program more simple, the programmer must be able to omit fields in the representation record for facts that he is not interested in.

2.3.3 Possibilities and limitations of Clean

Because we do not just want to derive types from the conceptual model, but also really use them for automated storage and retrieval, we need to take the possibilities and limitations of using the types in Clean into account. A simple choice may seem irrelevant for the programmer working with the types, but have a large impact on the design of the storage and retrieval mappings.

Real types instead of synonyms for identification

In the simple mapping algorithm we used type synonyms of `Int` and `String` to represent references to objects. This is a simple approach which provides the programmer with the possibility of referencing an object. While its simplicity may be attractive, it is too imprecise to use for generic mapping functions. The types we use for identification need a minimal amount of information about the entity they are referencing. Apart from the similarity in name there is no link between the entity types and their identification types. There is no relation between the reference types `Int` or `String`, and the entity it references.

Since an entity type can always be identified by a subset of its record's fields, the obvious choice for the identification type would be to use a record as well. This record contains just the fields that are required to identify an object. For example, the `EmployeeID` is then defined as follows:

```
:: EmployeeID   =   {employee_name :: String}
```

Another possibility which maintains the relation between the identification and entity types, is the use of "shadow types". These are types that do not use all of their type parameters in their constructors. Using a shadow type `ID` we can define identification types as follows:

```
:: ID a b       = ID b

:: EmployeeID   :== ID Employee String
```

A disadvantage of this shadow type based approach over using records is that they do not contain the information about the entities directly. While a link to the referenced type is available, the identification value is still just a string.

Naming conventions

The generic functions that work with these types, have to get all their information about the structure of the database from the type of an object. All extra information that we need to supply the generic functions with, has therefore be present in the type definition. A way to this is by using conventions in the naming of types and fields. The generic functions can then inspect these names and make case distinctions based on properties of them.

The initial approach already used the `ID` suffix convention to distinguish types representing entities and types representing references to entities. But to provide the necessary information to the generic mapping functions later on, we will use structured convention for naming record fields which allow us to encode the relations between entities in the types.

2.4 A second approach

With all the extra considerations we have seen in the previous section, we now specify the final mapping from ORM models to Clean data types. The types derived using this mapping are used for deriving a relational model and automatic storage and retrieval of these data types in the upcoming chapters.

2.4.1 The conceptual models

For reasons of simplicity, we continue working with just a subset of the ORM language. The mapping we define can therefore only be applied to 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 has roughly the same expressive power as the widely used Entity Relationship (ER) modeling language. The main difference is that in ER identification is not limited to single values.

While this subset ignores some of the more interesting aspects of ORM that make it a more expressive language than ER, it is still rich enough for modeling a large class of non trivial domains. Because the emphasis of this thesis is on the application of generic programming techniques we leave extension of the mapping to support the full ORM language to further research and just use this simplified version and its derived data types as the context of our generic functions.

2.4.2 The representation types

The types in Clean that are used to represent parts of the population are designed with conflicting objectives in mind. On the one hand, they should be easy to use by a programmer. Therefore they should be as simple as possible and require no unnecessary work to manipulate. On the other hand, they should contain *all* information required to map instances of these types to the database.

Entity records

Records are used as the primary construct to represent entity types in the ORM model. While the name of these record types may be freely chosen, it is advisable to use a the name of the entity type it represents. The names of the fields of a record have a strict 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.

- `<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>_<match 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 a record can have are limited as well. Fields in a record can either be a scalar type (`Int`,`Bool`,`Char`,`String`,`Real`), record type, a `Maybe` of scalar type or record, or list of scalar type or record.

While not immediately visible in the type definition, there is another property of the records representing entities that should be mentioned here. The first field of a record must be a one to one relation with a value type and is considered to be the identification value of that entity. This means that all instances of that type that have the same value in the first field, are considered to refer to the same entity in the database.

Identification records

To prevent endless recursion in the generic functions, we have to use references to entities instead of actual instances of these entity types at some point. For example, in many-to-many relationships between entities, only one of the entity records can contain the other entity. If both use the actual entity type, reading of one entity from the database causes reading of the related entity, which in turn requires the reading of the original entity again.

To prevent such cycles we define an ID record for each entity record that we define. This record type has the same name as the entity record with the suffix "`ID`". Identification records always contain just one field which has exactly the same name and type as their corresponding entity records.

Scalar types

Value types in the conceptual model are mapped to the basic scalar types in Clean. We assume that each value type can be represented by such a simple scalar. This assumption may not hold for real world applications, but keeps the mapping simple. It is trivial to increase the range of scalar types to which value types can be mapped, but this increases the number of base cases for the generic functions that perform the storage and retrieval and therefore increases the implementation effort.

Maybe values

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 have a value or may be empty.

Lists

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.

2.4.3 The mapping algorithm

With all the problems and considerations of the first approach in mind, and with the input and output domains of the mapping algorithm well defined in the previous sections, we now define our final algorithm:

1. Define *two* Clean record types for each object type in the ORM model. The first of the two records is used as *entity record* and its name must be equal to the name of the entity type in the ORM model. The second record is used as *identification record* and has the same name, but with an `ID` suffix.

2. In each of these records, (both the entity and identification records) add a field for the identification of that entity. This field must be of the form `<entity name>_<value name>` where `<entity name>` is the name of the record converted to all lower case characters. The `<value name>` must be the name of the primary identification of the ORM entity type. The type of the record field must be a scalar type which best fits the domain of the primary identification of the ORM entity type.

3. Map each fact type in the ORM model to fields in the set of entity records. The following cases can be distinguished:

   - If the fact type is unary, it is mapped to a field of the form `<entity name>_<value name>` in the entity record corresponding to the ORM entity type to which the fact is connected. `<entity name>` is the lowercase name of the entity record and `<value name>` may be freely chosen as long as it is unique in the entity record to which it is mapped. The type of the record field is `Bool`.

   - If the fact type is binary and there is a uniqueness constraint on only one of the roles, it is also mapped to a record field of the form `<entity name>_<value name>` in the entity record of the entity type connected to the role with the uniqueness constraint. The `<entity name>` is again the lowercase name of the entity and the `<value name>` can be freely chosen but has to be unique within the entity record.

     If the role without the uniqueness constraint is connected to a value type in the ORM model, and the role with the uniqueness constraint also has a total role constraint, the type of the field in the record is the scalar type corresponding with the domain of the value type. If there is no total role constraint the type of the field is the `Maybe` of the scalar type.

     If the role without the uniqueness constraint is connected to another entity type in the ORM model, and a total role constraint is present for the unique role, the type of the field is either the identification record type or entity record type of that entity type. Which of the two types is used may be chosen by the designer. The identification record type can always be used, but the entity record type may only be used when none of the fields in that entity record has the current record as its type (including types wrapped in lists or `Maybe`s), or a field indirectly contains a type which has a field which type is the current record. If there is no total role constraint, the type of the field is wrapped in a `Maybe` type.

     Finally, we add a field to the entity record of the entity type connected to the role without the uniqueness constraint. This field has the form `<entity name>_ofwhich_<match name>`. The `<entity name>` must be the same as that of the field added to the other entity and the `<match name>` should be equal to the `<value name>` of the field in the other entity. The type of this field is a list of identification record type or entity record type. Which of those two, may be chosen on the same conditions as the field in the other entity record.

- If the fact type is binary and both roles have a separate uniqueness constraint, one has to be chosen as the primary role, and the procedure of the previous bullet is applied as if only the primary role had a uniqueness constraint. The only thing we have to do differently is define the type of the `<entity name>_ofwhich_<match name>` field in the entity record of the non-primary role. Instead of a list, we now directly use the identifier or entity record type as the type of the field, since we know from the uniqueness constraint that there can be at most one instance.
- If the fact type is binary and there is a uniqueness constraint spanning both the roles of the fact type, we map this fact to two fields of the form `<relation name>_<match name>_ofwhich_<match name>`. One in each role of the fact type. For this we need to choose a globally unique `<relation name>` and two `<match name>`'s which do not have to be unique, but may not be equal. With these names we now add the fields to the two entity record types. The field names are the same, but with reversed `<match name>`'s. So the first entity record gets an extra field of the form `<relation name>_<match name a>_ofwhich_<matchname b>` and the second of the form `<relation name>_<match name b>_ofwhich_<match name a>`. The types of the fields are lists of the entity or identification record types of the "opposite" entity. Here, the condition remains that the entity record type may only be used if the entity record type in which we are adding a field is not yet part of the entity record type we want to use as type of the field.

Because the algorithm described above, has a lot of little details which may be difficult to extract from this rather verbose natural language description, it is schematically reformulated as a, somewhat informal, flow chart in figure 2.1.

To illustrate the process even further we redo the mapping of the project management ORM model of figure 1.2, which yields the following set of types:

```
:: 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)
```

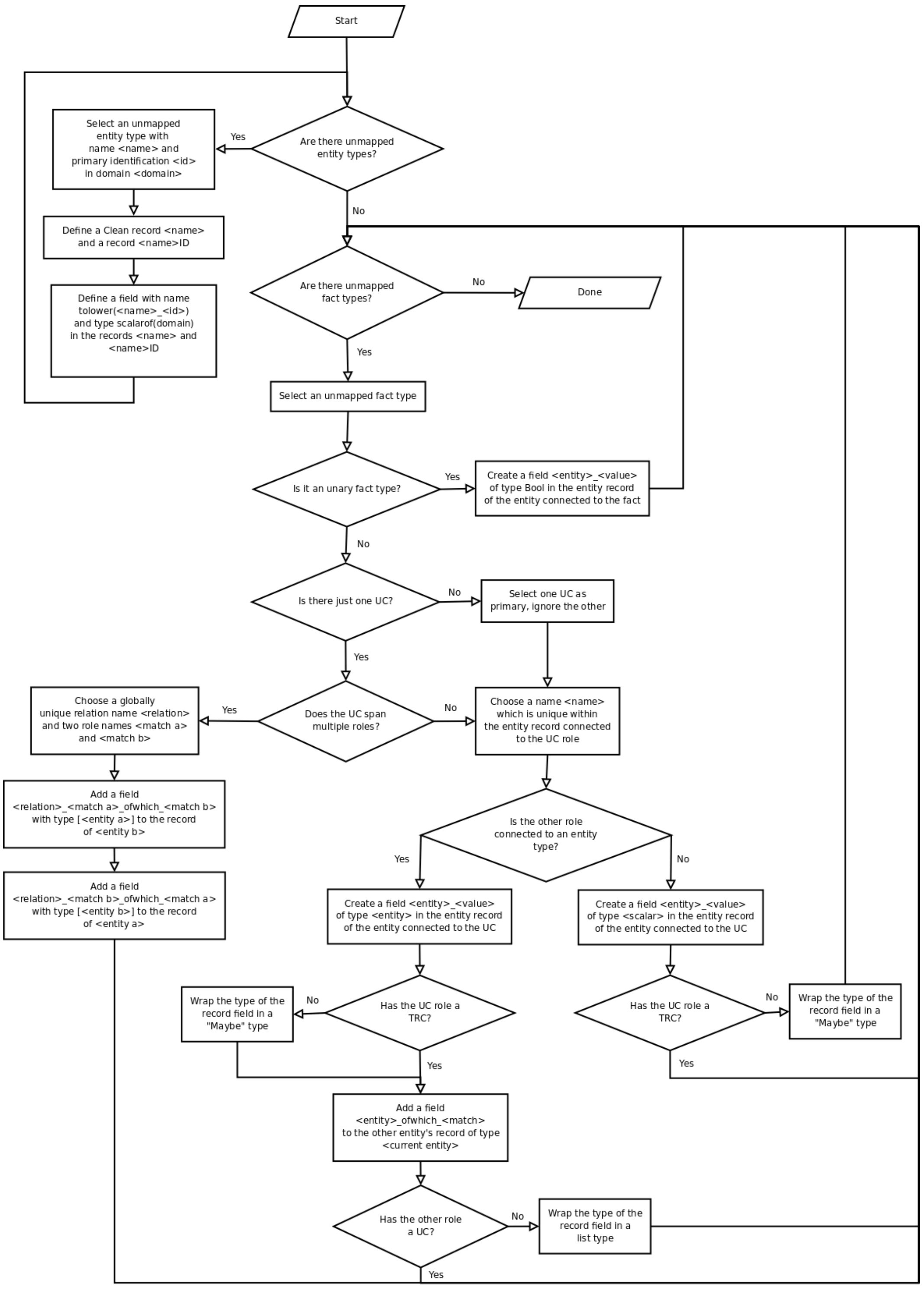

Start
Are there unmapped entity types?
Yes
Select an unmapped entity type with name <name> and primary identification <id> in domain <domain>
Define a Clean record <name> and a record <name>ID
Define a field with name tolower(<name>_<id>) and type scalarof(domain) in the records <name> and <name>ID
No
Are there unmapped fact types?
No
Done
Yes
Select an unmapped fact type
Is it an unary fact type?
Yes
Create a field <entity>_<value> of type Bool in the entity record of the entity connected to the fact
No
Is there just one UC?
No
Select one UC as primary, ignore the other
Yes
Does the UC span multiple roles?
Yes
Choose a globally unique relation name <relation> and two role names <match a> and <match b>
Add a field <relation>_<match a>_ofwhich_<match b> with type [<entity a>] to the record of <entity b>
Add a field <relation>_<match b>_ofwhich_<match a> with type [<entity b>] to the record of <entity a>
No
Choose a name <name> which is unique within the entity record connected to the UC role
Is the other role connected to an entity type?
Yes
Create a field <entity>_<value> of type <entity> in the entity record of the entity connected to the UC
No
Create a field <entity>_<value> of type <scalar> in the entity record of the entity connected to the UC
Has the UC role a TRC?
No
Wrap the type of the record field in a "Maybe" type
Yes
Has the UC role a TRC?
No
Wrap the type of the record field in a "Maybe" type
Yes
Add a field <entity>_ofwhich_<match> to the other entity's record of type <current entity>
Has the other role a UC?
No
Wrap the type of the record field in a list type
Yes

Figure 2.1: The mapping algorithm as flowchart

```
                , task_ofwhich_project                      :: [Task]
                , project_ofwhich_parent                    :: [Project]

                , projectworkers_employee_ofwhich_project   :: [Employee]
                }

:: ProjectID =  { project_projectNr                         :: Int
                }

:: Task =       { task_taskNr                               :: Int
                , task_project                              :: ProjectID
                , task_description                          :: String
                , task_done                                 :: Bool
                }

:: TaskID =     { task_taskNr                               :: Int
                }
```

When we compare this set of types with the set in our first approach we see two big differences. The first is the use of structured record field names, which now have an explicit relation to the conceptual model. The second is the use of records for the reference types instead of the simpler type synonyms.

Chapter 3

Mapping types to relational models

While the representation types in the previous chapter may be able to represent parts of a database in a Clean program, they have no connection with relational databases yet. In order to actually store and retrieve instances of these types in a relational database, there has to be a real database with tables in which the data is stored.

This chapter describes how we systematically derive a relational model from a set of representation types which we have derived using the algorithm in the previous chapter. This process is relatively simple because the mapping has been designed in such a way that the representation types contain all the information that is needed.

3.1 Information extraction from the types

The first step in the transformation of the representation types to a relational model is the extraction of relevant information from a set of types that has been derived from a conceptual model. The extraction of type information from the types is done in two passes in which all fields of all entity records are inspected. The first pass collects all information that is available without any context in a record field. The second pass collects some type information that in certain cases can only be determined with lookups in the information collected in the first pass.

In the first pass, we determine a set of properties for each field of each record in the set of representation types without the "ID" records. These are ignored because they are partial copies of the corresponding entity records and therefore contain no additional information. The properties we determine are:

- **Entity**
  To which entity record does this field belong.

- **Field**
  The field name in the record.

- **Relation type**
  The type of the record field where we ignore if a field is a `Maybe` value, a list or an "ID" version of the type. So the relation type of `Maybe EntityA`, `[EntityA]` or `EntityAID` is in all cases `EntityA`.

- **Scalar**
  Is the field of scalar type (`String`, `Int`, `Bool` etc.)?

- **List**
  Is the field of list type? E.g. `[EntityA]`.

- **Maybe**
  Is the field of `Maybe` type? E.g. `Maybe EntityA`.

- **ID**
  Is the field of "ID" type. E.g. `EntityAID`.

- **Key**
  Is the field the identifying key? The first field of a record is considered to be a key. The other fields are not.

- **Relation**
  The name of the relation to which we are mapping this field. This is the part of the field name before the first underscore. For example, the relation name of a field named `entitya_name` is `entitya`.

- **Select field**
  The name of the "select field". This is the name of the column in the database to which this record field is mapped. For field names without the `ofwhich` keyword, this is the part of the field name after the underscore. For fields with the `ofwhich` keyword this is the part of the field name after the relation name and before the `ofwhich` keyword. In some cases the `ofwhich` directly succeeds the relation name. In these cases the select field is undefined.

- **Match field**
  The name of the "match field". This is the name of the column in the database which is used to match on to select database rows during operations. For field names with the `ofwhich` keyword, this is the part after the `ofwhich`. For the other fields this property is undefined.

If we determine these properties for the types of the project management example of figure 1.2 which we have derived in section 2.4, we can obtain table 3.1 without the last two columns.

First pass											Second pass	
Entity	Field	Relation type	Scalar	List	Maybe	ID	Key	Relation	Select field	Match field	Select type	Match type
Employee	employee_name	String	Y	N	N	N	Y	employee	name	*undefined*	String	*undefined*
Employee	employee_description	String	Y	N	N	N	N	employee	description	*undefined*	String	*undefined*
Employee	projectworkers_project_ofwhich_employee	Project	N	N	N	Y	N	projectworkers	project	employee	Int	String
Project	project_projectNr	Int	Y	N	N	N	Y	project	projectNr	*undefined*	Int	*undefined*
Project	project_description	String	Y	N	N	N	N	project	description	*undefined*	String	*undefined*
Project	project_parent	Project	N	N	Y	Y	N	project	parent	*undefined*	Int	*undefined*
Project	task_ofwhich_project	Task	N	Y	N	N	N	task	*undefined*	project	*undefined*	Int
Project	project_ofwhich_parent	Project	N	Y	N	N	N	project	*undefined*	parent	*undefined*	Int
Project	projectworkers_employee_ofwhich_project	Employee	N	Y	N	N	N	projectworkers	employee	project	String	Int
Task	task_taskNr	Int	Y	N	N	N	Y	task	taskNr	*undefined*	Int	*undefined*
Task	task_project	Project	N	N	N	Y	N	task	project	*undefined*	Int	*undefined*
Task	task_description	String	Y	N	N	N	N	task	description	*undefined*	String	*undefined*
Task	task_done	Bool	Y	N	N	N	N	task	done	*undefined*	Bool	*undefined*

Table 3.1: Information extracted from the representation types

The last two columns of table 3.1 are determined during the second pass of the information extraction phase. The properties in these columns are the types of the select and match fields and are determined using the information from the first pass in the following way.

- **Select type**
  If the select field is undefined, then the select type is obviously undefined as well. In other cases the select type depends on the relation type. If the record field is a scalar, then the select type is the same as the relation type. When the field is not a scalar, the select type is the type of the key field of the relation type entity. We can find this type by searching the table for the row where the "Entity" property equals the "Relation type" property of the current field, and the "Key" property is "Y" ("yes"). There should be exactly one such row in the table. The select type of the current field is then the relation type of this row.

- **Match type**
  This type is determined similar to the select type. Again, when the match field is undefined the match type is also undefined. When the match field is defined, the match type is equal to the relation type of the key field of the current entity. This type can be found by searching the table for the row where the "Entity" property equals the "Entity" property of the current field, and the "Key" property is "Y". The match type is then the value of the "Relation type" property of that row.

3.2 Derivation of the relations

With all the information extracted from the types organized in a single table, the actual derivation of the relational model is pretty simple. For each combination

Relation	Field	Type	Null
employee	name	String	N
	description	String	N
projectworkers	project	Int	N
	employee	String	N
project	projectNr	Int	N
	description	String	N
	parent	Int	Y
task	taskNr	Int	N
	project	Int	N
	description	String	N
	done	Bool	N

Table 3.2: The relations derived from the representation types

of the "Relation" and "Select" properties, we define a field (column). The name of that field is the value of the "Select" property of that row in a relation (table) which name is the value of the "Relation" property. The type of this field in the relation is the value of the "Select type" property. If the "Maybe" property of the select/relation combination is "Y", null values are allowed. When the "Maybe" property is "N", null values are not allowed. If we repeat this procedure for the combinations of the "Relation" and "Match" properties and remove double occurrences of fields in the relations, we have the complete set of relations in which instances of the types can be stored.

Table 3.2 shows the relations that are derived from the information in table 3.1.

3.3 Derivation of the key constraints

The final step we need to complete the relational model is the derivation of primary and foreign key constraints. Primary keys are mandatory in most databases and describe which columns are used to uniquely identify records in a table. Foreign keys enforce that a record with a certain value in some table exists when that value also exists in another table. This restriction is used to prevent the database equivalent of a "dangling pointer".

Each relation has one and only one primary key. We can find which fields are part of the primary key of a relation by looking in the type information table and search for a row which "Relation" property is the relation we want to know the primary key of, and the "Key" property is "Y". If such a row exists, the primary key consists of one field which is the value of the "Select" property in the information table. If such a row cannot be found, the primary key consists of all the fields in the relation.

Table 3.3 shows the primary keys that have been derived from the information in table 3.1.

The foreign keys are derived from the type information table a little different. We look up up all the rows in the information table where the "Scalar" property is

Relation	Key fields
employee	name
projectworkers	project, employee
project	projectNr
task	taskNr

Table 3.3: The derived primary key constraints

Relation	Key field	Reference relation	Reference field
projectworkers	project	project	projectNr
projectworkers	employee	employee	name
project	parent	project	projectNr
task	project	project	projectNr

Table 3.4: The derived foreign key constraints

"N" and the "Select field" property is not undefined. With each of these rows we define a foreign key in the following way:

- **Relation**
  The relation on which the foreign key is defined is the value of the "Relation" property of the row.
- **Field**
  The field which is constraint by the foreign key is the value of the "Select" property of the row.
- **Reference relation**
  To find the reference relation of the foreign key we need to do an extra lookup in the information table. The reference relation is the value of the "Relation" property of the row in the table of which the "Relation" property equals the "Relation type" property of the current row and the "Key" property is "Y".
- **Reference field**
  The reference field is the value of the "Select" property of the key row we looked up to find the reference relation.

Table 3.4 shows the foreign keys that have been derived from the type information in table 3.1.

With three tables containing the relations (table 3.2), the primary keys (table 3.3) and foreign keys (table 3.4), the generation of SQL `CREATE TABLE` statements is a trivial exercise. One can simply iterate over all relations, lookup the primary key and the optional foreign keys for each relation and create the SQL statement.

Chapter 4

Mapping operations to SQL

With the mapping of conceptual models to relational models via Clean types, we have gained very little yet. We can now derive a set of data types that has a clearly defined mapping to a relational model, which we can derive from that set of types, but without functionality. To take advantage of the types we have derived as an intermediate stage in mapping to the relational model, we need operations on these types to allow storage and retrieval of their instances.

In this section we discuss a set of generic algorithms which implement the basic CRUD (create,read,update,delete) operations, for the types we have derived in chapter 2. With these operations available for each of our types, we can implement a rudimentary information system without writing any database access code.

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

4.1 The read operation

Before we can manipulate any data in a database, we first need to read that data from the database into our Clean data structures. For the types that we have derived in the previous chapters, we know that they can be mapped to a relational database, since we have derived their relational schema from the types themselves. Because the database schema is derived from the types, we know the relation between the types and the database, and can write an algorithm which reads instances of our types from the database.

Reading a data structure from a database is very similar to the parsing of a string or file. In both cases, a tree like structure is created from a serialized list of tokens, but instead of reading characters, we parse a list of values that have been fetched from a database. A big difference however, is that instead of having the input stream completely available when we start parsing, as we do with parsing a string or a file, we can only read 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. In order to find all these values in the various tables in the database we need information about the *type* of the (sub) data structure we are constructing, which is only available during parsing. Because of this, we cannot simply first read all values and then parse it into a data structure, but need to do a combined read and parse operation in which what is read next from the database, is guided by the parse process.

The token stream that we use to build our values can be viewed as a concatenated list of database rows, and therefore a list of different type of values. The actual token type we use does not only have constructors for value tokens which hold one `SQLValue`[1] value, but also special tokens such as list terminators which are required to guide the parsing process.

As we have seen in the chapter 2, the building blocks of our representation types are scalar values, records, lists and `Maybe`s. The rest of this section shows how each of these building blocks is constructed from a list of tokens.

4.1.1 Constructing scalar values

The easiest part of the read operation is the construction of scalar values. To create a scalar value, we simply have to examine the head of the token list. If it is an appropriate `SQLValue` token, we can use the value to construct our output value. If it is not, the parser fails. When the construction succeeds we remove the token from the head of the list.

For example, if we need to construct an `Int` and the head of the token list is `SQLVInteger` 42 we return 42. But if we need to construct an `Int` and the head of token list is `SQLVReal 3.03`, we are not able to create an `Int` and fail.

For the `Maybe` versions of scalar fields we do almost the same. We look at the head of the token list to see if it is `SQLVNull`. In this case we yield `Nothing`. If the head is an appropriate value to construct the scalar we yield `Just` that value.

4.1.2 Constructing 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.

[1]See appendix A for a complete reference of the SQL library

To construct a record, we obviously need to recursively construct all of its fields. But, we cannot demand that all the tokens needed to recursively construct the fields, are already in the stream. This is because we start the read operation with an empty stream and read tokens as we go along. Luckily, we do not need to have the tokens for each field in the stream. Because, in our representation types, the first field of a record is always considered to be a unique identifier, we can look up the tokens for the other fields when we only have the token for this key field.

The tokens for a field in a record are found by creating and executing an SQL query which retrieves these tokens. Because of the design of our representation types, we have all the information we need to construct such a query available encoded in the type of the record we are constructing. Depending on the type and name of the field we need different pieces of information to create the query for that field. The cases we distinguish are the same as those we have seen in the description of the entity records in section 2.4.2.

- `<entity name>_<value name>`
  For fields of this form the SQL query is very simple. Since the key field is always of this form, we simply look up a record by matching on the column of the id field. The query we generate has the form:
  ```
  SELECT <value name>
  FROM <entity name>
  WHERE <value name of key> = <value of key>
  ```
  For example:
  `SELECT description FROM project WHERE projectNr = 2.`

- `<entity name>_ofwhich_<match name>`
  Fields of this form are a little more complex. In this case we know the column to match and the table from the field name, but we do not know which column we should select. To obtain this column, we need to look at the type of the field. If it is a record, we select the `<value name>` part of the key field of that record. If it is a list or a maybe, we use the key field of the record inside the list or `Maybe`. Thus, the query will be of the following form:
  ```
  SELECT <value name of key of nested record>
  FROM <entity name>
  WHERE <match name> = <value of key>
  ```
  For example:
  `SELECT taskNr FROM task WHERE project = 2.`

  Note that the tokens we read for this field, are nothing more than the identification values of the entities or reference in that field. Reading of the entities themselves does not happen until the read operation is applied recursively.

- `<relation name>_<match name>_ofwhich_<match name>`
  Fields of this form are easier again since we have all the information available in the field name. The only extra information we need is the value of the

current record's key field. The query is then of the form:

```
SELECT <first match name>
FROM <relation name>
WHERE <second match name> = <value of key>.
```

For example:

```
SELECT employee FROM projectworker WHERE project = 2.
```

In this case, the query also just gives us the identification values of the entities or references.

After execution of the query, it depends on the type of the record field how we deal with the result returned from the database. If the type of the field is a list of entities or references, we retrieve all rows and add the values in the rows as value tokens to the stream. After that, a special terminator token is added to the token list to indicate the end of the result set. If the field is not a list, the result set consists of exactly one row and we can add the value in the row to the token list. Otherwise, an error is raised and the entire read operation fails. Note that in this case we do not need to add a terminator token because the amount of tokens needed to construct the field is fixed.

4.1.3 Constructing lists

The construction of lists is straightforward. As we have seen in the previous section, the tokens needed to construct a list are always terminated by a special terminator token. Therefore, when we want to construct a list of type `a` and the head of the token list is a terminator token, we remove that terminator from the token list and yield an empty list. If the head of the token list is not a terminator, we construct a list where the head is the result of constructing a value of type `a` with the token list, and the tail is the construction of a list of `a` with the rest of the token stream.

4.1.4 Initializing the token stream

The final step of the read operation is the initialization of the token stream. Because additional tokens are read at the construction of a record, we only need a token list with one value token to get the read operation going. This single value is the unique identifier of the data structure we want to read from the database. To create this token list, we do the inverse of the construction of scalar values. If we are given a certain scalar, we add the corresponding value token to the head of the list.

4.1.5 Putting it together

We now have all the parts needed to do a read operation for our representation types. Given a scalar value which identifies an entity we initialize a token stream and start constructing the entity. During the construction of the entity we consume the token stream, except during the construction of a record. In those cases, we infer from the types of the fields of that record where the data for construction of those fields can be found. We then fetch that data, and recursively construct the fields.

4.2 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. But the create operation is not an exact inverse of the read operation, because of some rather subtle details.

The first issue which complicates the create operation is the use of "auto increment" functionality of a database engine. This option, which is common in most database engines, is used to let the database assign automatically incrementing numbers to a key field when a new record is inserted. When the value `NULL` or `0` is used as the value of the key field in an `INSERT` statement, this value is replaced by a newly assigned number. After the statement has been executed, the value that has been inserted can be retrieved from the database. When multiple users concurrently create records in a database this feature is very useful because it guarantees that each record receives a unique key value.

If we want to allow the use of this feature in our representation types we need to be very careful in what order we execute the various `INSERT` statements that are needed to store our Clean data structures in the database. Since identification values are not known until a record has been created in the database, and that identification may be required for the creation of related entities, the order in which that entity and the related ones are created matters.

Another related issue which adds to the complexity of the operation is the existence of integrity constraints in the database. In chapter 3 we have not only derived the database tables from our representation types, but also a set of foreign key constraints. These constraints help to enforce the integrity of our data. But, these constraints also impose a certain order on the execution of `INSERT` statements because some records may only be created when another record already exists.

4.2.1 Creating scalar values

Just as in the read operation scalar values are only taken from the token list and not actually read from the database, during the create operation scalar values are not actually created in the database but merely accumulated in the token list. When an `Int` is "created" an `SQLVInteger` value token is appended to the token list.

For maybe values, an `SQLVNull` value token is added when the value is `Nothing`. If the value is `Just x` then `x` is created.

4.2.2 Creating records

When we apply the create operation to an entity record, a corresponding record is created in the database. Because of the constraints on the order in which we create the records in the database, the create operation consists of two recursive passes. Whether a field is handled in the first or the second pass depends on the "form" and type of the field. In the first pass, the create operation is applied recursively to a selection of the fields in the record. After this pass, the token list contains all values that are stored in the same table as the key field of the record. At this point a record in the database is created and the value of the key field of the current entity record is determined. If the value is zero, we check if an auto incrementing key value was assigned, if not we simply use the value we found in the entity record's key field. When the record in the database has been created we remove the "used" values from the token list and only leave the key value. The last thing we need to do is the second pass with the fields that were ignored during the first pass. Since for these fields it is necessary to know the key value of the current record, we add special "override" tokens to the front of the token stream that contain the identification value of the current record and the fields to which they have to be applied.

Depending on the form of a record field they are dealt with slightly different. We therefore conclude this description of the create operation with a more detailed explanation for the three different cases.

- `<entity name>_<value name>`
  Fields of this form are dealt with during the first pass. The create operation is applied recursively which yields the value token for this field in the token list. When the `INSERT` statement is executed after the first pass, the column named `<value name>` is assigned the value from this token. In some cases, the create is not applied recursively to get the value of the token. If at the front of the token list a special "override" token is found whose name equals the current field name, we add the value from this override token to the list instead of recursing. In this way the override tokens make it possible to pass key values from one record down the recursion to a sub data structure.

- `<entity name>_ofwhich_<match name>`
  Fields of this form are handled in the second pass. They have to be, because the key value of the current record is required to create fields of the form `<entity name>_<match name>` in the sub record(s) within the field. We achieve this by adding an override token for that field name to the front of the token list.

- `<relation name>_<match name>_ofwhich_<match name>`
  Record fields of this form are used for many-to-many relations which are mapped to a separate table. This table links the two entities in the relation. Because the relation is stored in a separate table we do not only need to create the "content" of the field in the database, but also link the "content" of the field to the current entity record. These fields are therefore also delayed until the second pass. At this point the identifier of the current entity record has been determined after the first pass. We now recursively apply the create operation to the field to get the identifier(s) for the other side of the relation. With both sides of the relation known, we link the entities by creating the "link record" using the following SQL statement:

  ```
  INSERT INTO <relation name>
  (<first match name>,<second match name>)
  VALUES (<value of field>, <value of key>)
  ```

4.2.3 Creating lists

The create operation for lists is also the inverse of the read operation on lists. In this case we are given the actual list and apply the read operation recursively to all of its elements. This leaves us with values for each element in the token list. To indicate the end of the field, we now add a terminator token to the token list.

4.2.4 Putting it together

The create operation reduces entity records to a single scalar value in the token list by first recursively applying the create operation to fields which can immediately be stored, followed by applying the create operation on the fields which need the identifier value of the current record. For fields of type `Maybe`, the create operation is only optionally applied and for lists the create operation is applied to all elements of the list. When the create operation is completed, the token list contains exactly one value: the identifier value of the entity record that was stored in the database. This value is finally converted to a "real" value by applying the read operation on the singleton token list.

4.3 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 we did before for the read and create operations, we cover the update operation for each of the building blocks of the representation types separately.

4.3.1 Updating scalar values

The update operation for scalar values is equal to that of the create operation. Because interaction with the database happens only during the processing of records, the update operation on scalars just turns values into tokens and nothing more.

4.3.2 Updating records

As with the read and create operations, the interesting part of the operation is the update of records. The update of a record is done in three passes, where the first and second pass are almost equal to the two passes of the create operation. Before the first pass, the fields of the record are read from the database. These are kept until after the second pass to determine of entities have been removed from list or maybe fields. After this read, the first pass recursively updates the fields which have to be stored in the record. When this is completed, the record in the database is updated with an SQL `UPDATE` statement. Because it is possible that the entity we are updating was a new entity added to a list, we need to check in the database if we have actually updated an existing database record. If this was not the case, we do an insert of that record as if we were doing a create operation. The second recursive pass is now be performed with the identification of the updated/inserted record passed along as override tokens. During this pass, the fields which refer to the current record in the database are updated. After the second pass, the link records are created in the database for those fields that represent many-to-many relations. When a link record already exists, no action is performed.

The last step of the update operation on records is a third recursive pass. This pass is only done in the update operation and garbage collects entities that have been removed from a list or a maybe field in the database. During this third pass, we compare for each field the new values we got from the first and second pass to the original values we read before the update. If the field is a list, we filter out the values that were present in the original list, but are no longer in the updated list. We then recursively apply the delete operation to those values. If the value is of type `Maybe` we check if the original value was a `Just` but the updated value is `Nothing`. In this case the delete operation is applied to the value in the `Just`.

As with the create operation, we are only interested in the record's identification value after the update is complete. This is therefore the only token that is added to the token stream.

4.3.3 Updating lists

Unsurprisingly, updating lists is also similar to creating lists. The update operation is applied to each member of the list which add their share to the token stream. After that a terminator token is added to the token list to indicate the end of the items.

4.4 The delete operation

The final operation we need, to complete our set of basic operations, is the delete operation. This operation is very similar to the read operation, but instead of just reading records from the database we remove them after we have read them. In order to delete all information about an entity we need to find everything in the database. Therefore, we can just as easily construct the entity we are deleting as a welcome side effect. The delete operation thus returns *almost* the same value as the read operation, but alter the database while it reads. The read data structure is *almost*, but not entirely the same as the result of the read operation. This is because the database does not delete all the records in the database that represent an entity at once, it does this step by step while traversing the data structure. This incremental nature causes the effect that when a record deep in the structure is read from the database, the relations of that record with entities higher up in the structure are already deleted. These relations are therefore not present in the constructed data structure.

For the last time, we now explain the delete operation for each of the building blocks of our representation types.

4.4.1 Deleting scalar values

The delete operation for scalar values is the same as the read operation for scalar values. The tokens that have been read during the processing of records are consumed to construct the data structure.

4.4.2 Deleting records

While deleting records is similar to reading records, there are once again some subtleties that need to be taken into account.

Because of the integrity constraints in the database, we cannot delete records when other records still reference them. Therefore we apply a multi-pass strategy again. Records that are not referenced by other records are read and then deleted immediately. Records who are referenced by others are first read, and only later deleted.

If the delete operation is applied on a record during the read or combined read-/delete pass, the values for this record are read first. After this the operation is applied recursively to the fields of the record. Depending on if a field references another entity, the recursive call does an only read pass or a combined read/delete pass. If the delete operation on the current record was during an only read pass, we are done. For a combined read/delete we need to delete the current record. Because we have the identification value of this entity in the token list, as we had with the read operation, and records in the database are uniquely identified by this key value we, can perform the deletion with the following SQL statement:

`DELETE FROM <entity name> WHERE <value name> = <key value>`.

The `<entity name>` and `<value name>` are determined from the name of the key field of the current record, which always has the form `<entity name>_<value name>`

When we have deleted the record from the database we can now perform a garbage collection pass to delete the fields which were only read during the first pass.

When a garbage collection pass of a delete operation is performed on a record, we assume that the tokens to construct the record are already in the token list, because they have been read during a previous pass, and construct the data structure. We then delete the record from the database in the same way we did as in the combined read/delete operation.

4.4.3 Deleting lists

As with the read operation on lists, we construct an empty list when the head of the token list is a terminator, and apply the operation recursively on the token list otherwise to construct the elements of the list.

4.5 Implementation of the operations in Clean

The operations described in the previous section outline the general recipe for reading, creating, updating and deleting entities in a database. But, they are not a working implementation yet. To show that the operations not only work in theory but can actually be used in a program, they have have been implemented in a Clean library. This library, called "GenSQL" implements the four operations by means of a generic function and a set of wrapper functions.

4.5.1 Basics first: A database API for Clean

Because the standard library of Clean has no support for working with databases and no suitable other libraries were available, the first step in the implementation was the development of a database API for Clean. This API has been designed based on the DB API 2 of Python [12], but adapted to fit the functional paradigm instead of the Object Oriented approach of the Python version. This library specifies and implements various general data types and functions for working with databases, and defines the functions for the actual interaction with a database. It can be implemented for various database backends. For this thesis only a backend for MySQL [14] has been implemented, but the API abstracts over the details of different database systems and other backends can be implemented easily.

4.5.2 Jack of all trades

A limitation of the generics mechanism in Clean is that it is not possible for generic functions to call other generic functions. In the design of the GenSQL library this was a severe limitation, because the various operations have some overlap in their functionality. The update operation, for instance, uses the delete operation during a garbage collect step. An even larger piece of functionality, is the extraction of type information about record fields, which happens in all the operations. Some of the operations also traverse the data structures in multiple passes. While these passes do different things, we are unable to define them as separate generic functions which call each other.

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. The optional error can have two causes, as expressed by the `GSQLError` type:

```
:: GSQLError = GSQLDatabaseError SQLError
             | GSQLTypeError String
```

The most likely error, is that something went wrong with the database. If this happens, the `SQLError` from the database is embedded in the `GSQLError`. The other thing that can cause an error, is the use of the `gSQL` function on values which type is not a representation type. Since generic functions by definition are defined for every type, we can only detect at runtime that a programmer applied the function to a type for which it was not meant. In these cases an `GSQLTypeError` is returned which contains a string explaining the error.

4.5.3 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.

4.5.4 Performance

By using the generic mapping, the programmer no longer has to do write the SQL queries to read or write in the database. The four operations are essentially a black box, which either take or yield data structures. *How* these structures are read or written, is out of the programmer's control. Because of this black box nature of the mapping, it is easy to forget that there is actually a lot work going on under the hood.

It is important to realize that the current operations are not optimized to be efficient in the amount of database interaction that is done to perform the operation. Because the mapping operations only minimally look ahead, a lot of information that could be retrieved or updated in one query is now spread out over multiple queries. The worst performance disadvantage that the current mapping suffers from, is the way it handles lists of related entities. For example when a `Project` is read, then in order to read the `Tasks` related to that `Project`, a query is done to get all the identification values of the related `Tasks`. Then the read operation recursively reads the tasks where for each `Task` a number of queries is done to fill the fields of the `Task` record. In this approach, the number of queries is linear in the amount of related `Tasks`. When the mapping was not used, one could retrieve all the `Task` information with a single query "`SELECT * FROM task WHERE project = ?`", thus the number of queries is constant in the amount of related `Tasks`. When a `Project` has a lot of `Tasks`, this is a serious performance issue, which puts a lot of extra stress on the database engine.

Another performance issue which is mostly noticeable in the memory consumption of applications using the mapping, is the overhead that is introduced by the use of generic functions. Because of the translation to and from the generic domain, generic functions always have some overhead in both memory and execution time when compared to normal functions. However, when optimization techniques [3] are applied in the compiler, this generic overhead can be completely removed. Unfortunately, the Clean compiler does not support this yet.

4.6 Intermezzo: An elegant way to share

The operations we introduced in this chapter are designed for information systems. In these systems the use of a relational database for storage is taken for granted. But when viewed in a different context, they can be seen as a solution for one of the more difficult issues in functional programs: sharing of data. Where it is easy in imperative or object oriented languages to create a series of data structures in memory which reference each other, and in that sense share data between those structures, this is harder to realize in a functional language.

Because the representation types contain just enough information to be able to manipulate an entity, which is related to other entities in the database, there is no need to create a data sharing structure in your Clean program. All sharing of data is handled in the database and is elegantly hidden from the functional programmer.

For applications that need a way to maintain some structure of related entities, the right combination of types and tables plus the generic operations might save a lot programming effort.

Chapter 5

Mapping relational models to types

Even though we now have every step to get from a conceptual schema to a collection of types, a corresponding relational schema, and a set of generic operations to automatically map between them, we still have an important question to answer: Can we, and if so, how can we, apply these results to existing databases?

Because there are many different methods for designing the database of an information system, it is unlikely that an arbitrary database was designed based on an ORM model using just the subset defined in this thesis. It is more likely that the database was designed using another modeling technique or even no modeling technique at all. It may even be the case that, while the database was originally designed using a formal conceptual modeling technique, the models have been lost, or have become outdated.

In these cases, we have no conceptual model to derive our types from. Fortunately, not all is lost. Because a relational model can always be extracted from a database itself, we can still use our generic functions if we can derive the types from this relational model instead.

In this chapter we see what properties a database must have in order to be used with our generic operations, and how we can derive the representation types from a relational model instead of a conceptual one.

5.1 Preconditions

Because our mapping operations have been designed under the assumption that the database to which they are applied has been derived from the representation types, we can only deal with databases that *could* have been derived from a set of representation types. In order to apply our mapping to an existing database, we need to be able to construct a set of representation types which, when we would apply the procedure of chapter 3, would yield that database.

5.1.1 Table and column names

The mapping operations determine table and column names based on the names of the record fields of the representation types. These names consists of several parts separated by underscore characters and the `ofwhich` keyword. The first part is used as table name, and the second and optionally third part indicate columns in the database. A consequence of this mandatory naming convention for record fields is that the names of tables and columns in an existing database are not allowed to contain any underscores. Another consequence is that a column can not have the name "ofwhich". If a table does not meet these requirements we are not able to map it to representation types.

The fact that we cannot use "ofwhich" as a column name, is a limitation that will not likely be a problem. The exclusion of underscores from table and column names is a bigger problem however. Because it not uncommon to use them to separate names consisting of multiple words, it is very likely that an existing database contains tables or columns with underscores. Luckily, there are no reasons other than readability why the parts in a record field are separated by an underscore. One could simply use another character or even a string of characters to separate the parts. In the current implementation of the mapping, one can change the separator string by simply changing a single macro definition in the library's definition file.

5.1.2 Table structure and keys

Not only are the names of tables and columns limited, but the structure of tables and the types of columns are somewhat limited as well. When the tables are derived for a set of representation records, there are two categories of tables. The first type of tables are tables that represent entities. In these tables all facts and relations that can be stored in a single value are collected for an entity. The first column of these tables is always the primary key and is used to identify the entities. The other type of tables are the link tables. These tables are used to represent many-to-many relations between entities and always consist of two columns containing the identifications of the entities involved in the relation. The primary key of these tables always spans both columns.

If we want to use our mapping with an existing database, we can only map the tables from this database that can be put in one of the two categories mentioned above. If a table is neither an entity record nor a link record, we cannot map it to a representation type.

The final precondition on the structure of the tables, is that the data types of columns is limited to the column types that are supported by the mapping. In the current implementation of the mapping we are limited to the column types `CHAR`, `VARCHAR` and `INTEGER` which are used to map the Clean types `Char`, `Int`, `Bool` and `String`. This limitation however is merely a practical one. It is trivial to extend the mapping to include other scalar types.

5.1.3 Relation between tables

In the representation types, the relation between entities can be inferred from the types of the record fields. These relations are enforced with foreign keys in the derived database tables. In existing databases there do not need to be any foreign key constraints. This does not mean however, that there are no relations between tables. It is very well possible, and even likely, that some columns do not represent attributes of an entity, but refer to the primary key or another entity. If there are no foreign keys, we cannot know to which other table the values in such a column refer. If we want to use the full potential of our mapping, we need to know which entities are related to determine the types of the representation type record fields. So the database we want to use has to either have foreign keys from which we infer the relation between entities or another source of knowledge about the relations in the database has to be available. Such a source could be informal documentation of the existing system or an interview with a maintainer of the system.

5.2 The mapping algorithm

If the database we want to use fulfills the preconditions defined in the previous chapter, we are able to derive the representation types from its relational schema using the following algorithm:

1. Determine which tables represent entities, and which ones are link tables representing relations between entities.
2. For each entity table define an entity record type and add fields to the record for each column in the table.
   - The name of the record type is the capitalized table name
   - The names of the fields in the records are of the form `<table name>_<column name>`
   - The types of the field are the Clean scalar types that correspond to the SQL data types of the columns.
3. Create an identification record for each entity record by defining a new record type which has one field which has the same name and type as the entity record. The name of the identification record is that of the entity record with an `ID` suffix.

4. For each field in the entity records of which we know it is a reference to another entity, change the type of the field from the scalar type to the identification record type of the entity that is referred to. Whether a field references another entity can be either inferred from a foreign key constraint on the table, or external knowledge about the database.

5. For each entity that is referenced by another entity, add fields of the form `<entity name>_ofwhich_<field name>` to the entity record where `<entity name>` is the entity type that refers to this entity type, and `<field name>` is the column name of the field in that entity record that references this entity type. The type of these fields is list of the identification record type of the entity that references this entity.

6. For each link table determine the entities that are referenced by each column in the table. Create a field of the form `<table name>_<column name>_ofwhich_<column name>` in the entity type records of both the entities involved in the relation. `<table name>` is the name of the link table and the two `<column name>` fields are the names of the columns in the link table where the last is the name of the column referencing the entity to which we are adding the field. The type of the field is list of the identification type record of the entity referenced by the first column.

7. The final optional step is the replacement of references by includes of entities. Each identification record type may be replaced by an entity representation type as long as no inclusion cycles are introduced. That is, the current entity (in which we are changing a reference to an include), may not be directly or indirectly included in the entity record type we want to include.

Just as we have done for the mapping algorithm based on conceptual models in chapter 2, we illustrate the algorithm by means of an informal flow chart in figure 5.1.

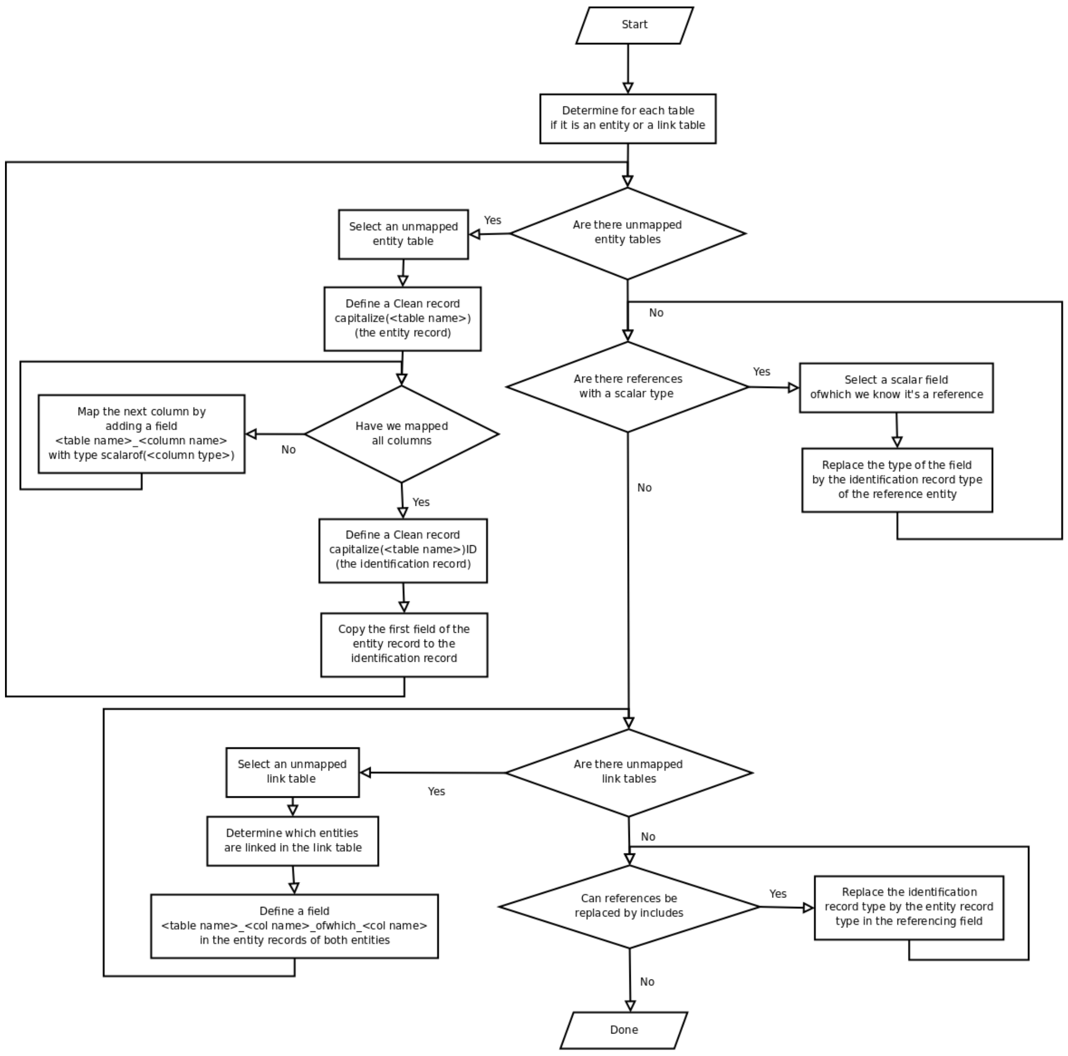

Start
Determine for each table if it is an entity or a link table
Are there unmapped entity tables
Yes
Select an unmapped entity table
Define a Clean record capitalize(<table name>) (the entity record)
Have we mapped all columns
No
Map the next column by adding a field <table name>_<column name> with type scalarof(<column type>)
Yes
Define a Clean record capitalize(<table name>)ID (the identification record)
Copy the first field of the entity record to the identification record
No
Are there references with a scalar type
Yes
Select a scalar field ofwhich we know it's a reference
Replace the type of the field by the identification record type of the reference entity
No
Are there unmapped link tables
Yes
Select an unmapped link table
Determine which entities are linked in the link table
Define a field <table name>_<col name>_ofwhich_<col name> in the entity records of both entities
No
Can references be replaced by includes
Yes
Replace the identification record type by the entity record type in the referencing field
No
Done

Figure 5.1: The mapping algorithm as flowchart

Chapter 6

Example: A Project Management System

To demonstrate the power of the generic mapping operations and to test the implementation of the GenSQL library, the project management system introduced in chapter 1 has been implemented. This small web based information system is built on a MySQL database and a set of representation types that are derived using the methods of chapters 2 and 3.

6.1 Application Design

The project management application is designed as CGI web application. A single Clean program is run each time an HTTP request is made and generates a response. Because of the web based approach, the application has been structured as a collection of pages. Each page is implemented as a function, which given a request and a database cursor, generates the title and content of a page. When the program is run, it parses the request, determines what page function should be evaluated, and finally wraps the result of that function in a layout and outputs a response.

In the application, only two of the three entity types are used as "entry points": projects and employees. You can only browse a list of projects and a list of employees, but not a list of tasks. This is because tasks are always related to exactly one project and have no meaning outside the context of a project. They are therefore included in the "show" and "edit" pages of projects.

The url that is requested determines which page function is executed. Figure 6.1 shows a tree structure of the mapping between urls and page functions. The two main branches are "/projects" and "/employees" which have pages for the basic operations beneath them.

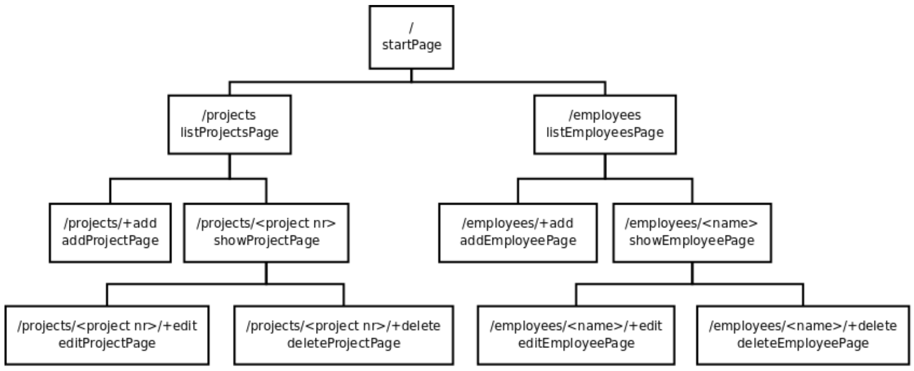


Figure 6.1: The page tree of the project management application

The user interface in this system is defined by hand for each page. For both projects and employees there is a function which generates an HTML form from a `Project` or `Employee` data structure, and a function which parses the result of submitting that form to a data structure again.

6.2 Types and Tables

The types and tables in this implementation differ little from those we have already seen in the examples in chapters 2 and 3. We start again with the ORM model in figure 1.2 and apply the procedure from chapter 2. This time we are a little more careful with the replacement of references by includes. Because we want to link employees to projects and vice versa, but do not want that projects are deleted when they are removed from an employee data structure, or only new employees can be added to a project instead of existing ones, we use references in the `projectworkers` relation. Tasks, on the other hand, are always related to a single project and are therefore safely included in the `Project` type.

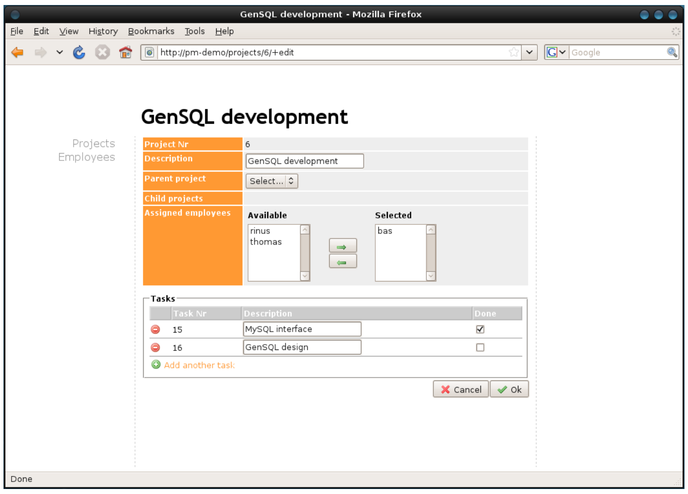
When we make these choices during the derivation, we get the following set of representation types:

```
:: 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]
```


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

Cancel Ok

Figure 6.2: Screenshot of the project edit page

```
              , 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
              }
```

Since the choice of using references or inclusions does not affect the database that is derived from these types, we can apply the procedure of chapter 3 again to get the same tables and primary and foreign keys as shown in tables 3.2, 3.3 and 3.4.

6.3 The mapping in action

A nice example where we see the GenSQL mapping in action in this system is the "edit" page of projects which is shown in figure 6.2. This page is implemented by the following Clean 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.

The complexity of the update operation on projects becomes clear when we look at a hand written update function for projects:

```
updateProject :: Project !*cur → (Maybe SQLError, *cur) | SQLCursor cur
updateProject project =: {Project | project_projectNr = pid} cursor
    //Update the project record
    # (mbErr,cursor)        = sql_execute
        "UPDATE project SET description = ?, parent = ? WHERE projectNr = ?"
    pvalues cursor | isJust mbErr          = (mbErr, cursor)
    //Update/create the linked employees
    # (mbErr, ids, cursor)  = linkEmployees
        project.projectworkers_employee_ofwhich_project cursor
    | isJust mbErr          = (mbErr, cursor)
    //Garbage collect linked employees
    # (mbErr,cursor)        = sql_execute
        ("DELETE FROM projectworkers WHERE project = ?" +++ ematch ids)
        (evalues ids) cursor
    | isJust mbErr          = (mbErr, cursor)
    //Update/add the tasks
    # (mbErr,ids,cursor)    = updateTasks project.task_ofwhich_project cursor
    | isJust mbErr          = (mbErr, cursor)
    //Garbage collect tasks
    # (mbErr,cursor)        = sql_execute
    ("DELETE FROM task WHERE project = ?" +++ tmatch ids) (tvalues ids) cursor
    | isJust mbErr          = (mbErr, cursor)
    = (Nothing, cursor)
where
    pvalues = [SQLVVarchar project.project_description
              ,pparent project.project_parent
              , SQLVInteger project.Project.project_projectNr]
    pparent Nothing = SQLVNull
    pparent (Just {ProjectID| project_projectNr = x}) = SQLVInteger x

    linkEmployees [] cursor = (Nothing, [], cursor)
    linkEmployees [{EmployeeID | employee_name = e}:es] cursor
        # (mbErr, cursor)   = sql_execute
            "SELECT * FROM projectworkers WHERE project = ? AND employee = ?"
            [SQLVInteger pid, SQLVVarchar e] cursor
        | isJust mbErr      = (mbErr,[],cursor)
        # (mbErr, num, cursor) = sql_numRows cursor
        | num == 0
            # (mbErr, cursor)       = sql_execute
                "INSERT INTO projectworkers (project,employee) VALUES (?,?)"
                [SQLVInteger pid, SQLVVarchar e] cursor
            | isJust mbErr          = (mbErr,[],cursor)
            # (mbErr,ids,cursor)    = linkEmployees es cursor
            = (mbErr,[e:ids],cursor)
        | otherwise
            # (mbErr,ids,cursor)    = linkEmployees es cursor
            = (mbErr,[e:ids],cursor)

    ematch []   = ""
    ematch ids  = " AND NOT (employee IN (" +++
                  (text_join "," ["?" \\ x ← ids]) +++ "))"
    evalues ids = [SQLVInteger pid: map SQLVVarchar ids]
```

```
updateTasks [] cursor = (Nothing, [], cursor)
updateTasks
    [{Task|task_taskNr = taskNr,task_description=description,task_done = done}:ts]
    cursor
    | taskNr == 0
        # vals                  =
            [SQLVVarchar description, SQLVInteger (if done 1 0)
            ,SQLVInteger pid]
        # (mbErr, cursor)       = sql_execute
            "INSERT INTO task (description,done,project) VALUES (?,?,?)" vals
            cursor
        | isJust mbErr          = (mbErr, [], cursor)
        # (mbErr, i, cursor)    = sql_insertId cursor
        | isJust mbErr          = (mbErr, [], cursor)
        # (mbErr, ids, cursor)  = updateTasks ts cursor
        = (mbErr, [i:ids], cursor)
    | otherwise
        # vals                  =
            [SQLVVarchar description,SQLVInteger (if done 1 0)
            ,SQLVInteger pid,SQLVInteger taskNr]
        # (mbErr, cursor)       = sql_execute
            "UPDATE task SET description = ?, done = ?, project = ? WHERE taskNr = ? "
            vals cursor
        | isJust mbErr          = (mbErr, [], cursor)
        # (mbErr, ids, cursor)  = updateTasks ts cursor
        = (mbErr, [taskNr:ids], cursor)

tmatch []   = ""
tmatch ids  = " AND NOT (taskNr IN (" +++
              (text_join "," ["?" \\ x <- ids]) +++ "))"
tvalues ids = map SQLVInteger [pid:ids]
```

This function is not very difficult to write and is not really special, but it is rather long. It is also written specifically for this data type and the related database tables. Without the generic mapping, we would need to write eight of these functions. One for each CRUD operation of both projects and employees. Even for such a small system, this is a lot of code.

Chapter 7

Conclusions

Now that we have seen that it is possible to apply generic programming to automate the mapping between data structures and databases, it is time to take a step back and reflect on what we have accomplished. In this final chapter we evaluate the results of the project to see what we have contributed, and ponder upon some of the future challenges that may spring from this project.

7.1 Evaluation

First, let us take a critical look at the mapping approach we have developed so far. To do so, we first look at the mapping from the viewpoints of some different quality criteria. We then look at how our mapping compares to the related work mentioned in chapter 1.

7.1.1 Applicability

We start with looking at the applicability of our work. By this we mean the range of problems to which the mapping can be applied. For our current mapping approach and implementation we divide this range into two categories: As basis for new information systems specified by an ORM model, and existing databases for which we want new interfaces.

The first area where we can use our approach, is in the construction of new information systems. If the universe of discourse is simple enough to be captured by the ORM subset we defined, our approach provides a way to systematically derive both the database and the core data types of the system from an ORM model. If they do so, they do not need to program any database interaction code, because all database input and output is handled by the type driven generic function. This might prove to be a good incentive to encourage developers to first make a conceptual model of a system instead of immediately start building databases and writing code.

The second way in which our mapping can be used, is by using the representation types as a view on an existing database. If an existing database meets the preconditions defined in section 5.1, we can define types based on the relational model of the database which reflect the entities stored in the database. The generic functions are then used to propagate changes in the views, the representation types actually, to the database. This alternative use of the mapping makes it possible to reduce the amount of database interaction code, even for systems that have to work with an existing database.

7.1.2 Usability

A more vague quality attribute of our mapping is usability. This attribute is usually used to "measure" how easy it is for end users of a system to use it. In our case, the users of the mapping are not the end users of an information system, but the designers and developers of the system. The aim of the mapping was to relieve programmers of a lot of repetitive and error prone work. Therefore the work one has to do when the mapping is used should be less than without the mapping. For a typical update of an entity in the database, the work one has to do could be summarized as follows:

Without the mapping:

- Do all SQL queries to retrieve the information about the entity
- Convert the returned lists of SQL values to a convenient data structure
- Manipulate the data structure
- Do all SQL queries to propagate the changes in the data structure to the database

With the mapping:

- Call `gsql_read` to retrieve all information about the entity in the form of a representation data structure for that entity.
- Manipulate the data structure
- Call `gsql_update` to propagate the changes to the database.

If we compare these two lists, we see that the programmer no longer has to do SQL queries anymore and that there is no need to do a conversion from the results returned by the database to a convenient data structure. The obvious advantage is that a programmer no longer needs to write SQL. In fact, he no longer even needs to know SQL. Not having to convert between the flat database results and a more convenient structure also saves a lot of boring work.

An additional advantage of the mapping is that it helps to prevent programming errors. Because the mapping executes all SQL statements that are necessary to propagate changes in the data structures to the database, it is impossible to make programming errors in an SQL statement, or even forget to execute a statement at all. Because of the repetitive nature of this programming work, it is very likely that a human programmer makes such mistakes every once in a while. The mapping therefore also saves debugging and testing effort and helps to protect the integrity of the database.

But what is the price we have to pay for these benefits? The first thing that we lose is control over the SQL queries that are executed. It is no longer possible to use clever queries that get the most information in as little queries as possible. As have seen in chapter 4, the current implementation is not the most efficient one, in terms of performance. But it is possible to optimize the mapping to get better performance. The second price the programmer has to pay, is that he is no longer free to choose a data structure to represent an entity. The data types used to represent entities are defined by the mapping and have to follow the rules of the mapping. The biggest disadvantage here is that the names of record fields can become annoyingly long. For example, updating a record field with the name `projectworkers_employee_ofwhich_project` means a lot of typing and long lines of code in your programs.

7.1.3 Generic mapping vs. object-relational mapping

To anyone that is familiar with object-relational mapping systems, such as for example Hibernate [9] in Java, the generic mapping presented in this thesis will appear very similar. On a certain level this true, because from a programmer's perspective they both serve the same purpose. Namely, the automation of SQL programming work to deal with the storage and persistence aspects of a system. From that point of view one could argue that the generic mapping is the functional language equivalent of object-relational mapping.

However, there is a subtle but important difference between the two approaches. This difference is the focus on what is mapped to what. In object-relational mapping, the object model is mapped to the relational model to make objects persistent. From a design perspective, this means that *objects* remain the key abstraction on which systems are built. The world is modeled in terms of objects, and some of these objects can be stored and retrieved from a relational database. In our generic mapping we do not try to map all abstract data types to a representation as a relational model, nor do we try to just map a relational model of a database to abstract data types. Instead, we realize that in an information system the world can be modeled on a conceptual level in terms of objects having relations which can be expressed as facts. In order to implement a system to record these facts, we need a relational model in which conceptual entities and relations can be stored, as well as a representation as data structure to manipulate them. Both of these representations are "shadows" of the same conceptual entities and can thus be mapped. Thus, the mapping we introduced does not map *all* types to a relational model, nor does it map *all* relational models to types. Instead it provides *both* a storage and a manipulation representation of conceptual entities and enables transparent mapping between these two representations.

7.1.4 Generic mapping vs. type safe SQL

The other existing technique which shares some goals with our generic mapping is the embedding of SQL inside a functional language to enable type safe expression of SQL queries and statements. The similarity is in this case that both approaches hide the "low-level" string manipulation of creating SQL statements from the programmer. The big difference here is that the embedding of SQL, while very useful to detect errors at compile time, stays on the level of relations and tables. Our mapping approach on the other hand, provides representations that reflect the conceptual structure of entities.

For the four CRUD operations, the need for compile time checking of SQL queries disappears because they are generated by our generic functions. For queries, which are not covered by our mapping, type safe SQL could complement our mapping. If a type safe SQL library for Clean had been available, it could have even been possible to built the mapping on top of it.

7.2 Contributions

The main contribution of this thesis is that it shows that given the right choice of types and database layout, it is possible to use generic programming to automate the mapping between entities stored in a database and their representation as data structures in Clean.

In order to do so we have shifted the focus away from both the database and the data types, towards the conceptual level of ORM models. Thereby at the same time introducing a way to use ORM models not only as basis for the design of databases, but for the design of the data types of programs that work with these databases as well.

The implementation of the mapping should be considered a proof-of-concept. Because of the limited applicability and the performance issues outlined in the previous section, it is not ready for production systems, but might prove to be a valuable tool for rapid prototyping.

7.3 Future research

In this thesis we have only scratched the surface of what is possible in the automation of the construction of information systems. Therefore we conclude this thesis with some suggestions for further work, that is inspired by the work on this project.

7.3.1 Improvement of the mapping

Since the state of the mapping at this point is that of proof-of-concept, a lot of further research can be done on improving the approach. The obvious issues are an extension of the ORM subset that can be handled, and improvement of the performance of the implementation. But one could also think of extension of the scope of the mapping. The current mapping focuses on operations on single entities. It would be interesting to extend this to operations on sets of entities. For example, the automatic selection of the set of entities available in the database that are candidates for a certain relation. Another interesting improvement is the possibility of mixing references and includes of related entities. In this way one could select both existing entities by reference, and add new ones by value during a single update of an entity.

7.3.2 Beyond data models

In this project we focused on conceptual data models to use as the basis for automating a part of the work involved in constructing an information system. These models, however, are just a part of the specification of an information system. Interesting further research is therefore the integration of our mapping with other generic approaches that focus on other aspects of an information system, such as the user interface [17] or work flow [18] of the system, to work towards fully model based development of information systems.

Appendix A

The Clean Database API

The standard library of Clean does not have any support for working with relational databases. There were some partial solutions available as part of other projects, such as a very basic ODBC interface as part of the iData framework [17], but no general purpose database API which abstracts over the specific database engines available.

This library, which is inspired by the Python database API version 2 [12], provides a set of data types and functions for working with relational databases and defines a set of type classes that define the functions a database library must implement.

Basic Types

The first part of the database API defines the types which are used in interaction with a database. Because the result rows of a database query often consist of fields of different types, the `SQLValue` type is defined which wraps the scalar data types found in various databases. Result rows are lists of `SQLValue` and statements are just plain strings.

```
:: SQLStatement     :== String

:: SQLValue          =   SQLVChar         String
                     |   SQLVVarchar      String
                     |   SQLVText         String
                     |   SQLVInteger      Int
                     |   SQLVReal         Real
                     |   SQLVFloat        Real
                     |   SQLVDouble       Real
                     |   SQLVDate         Date
                     |   SQLVTime         Time
                     |   SQLVTimestamp    Int
                     |   SQLVDatetime     Date Time
```

```
                    |   SQLVEnum        String
                    |   SQLVNull
                    |   SQLVUnknown     String

:: SQLRow           :== [SQLValue]
```

Errors

In interaction with databases a lot can go wrong. To capture these problems and adequately deal with them, the API defines an `SQLError` type.

```
:: SQLError = SQLWarning            Int String
            | SQLInterfaceError     Int String
            | SQLDatabaseError      Int String
            | SQLDataError          Int String
            | SQLOperationalError   Int String
            | SQLIntegrityError     Int String
            | SQLInternalError      Int String
            | SQLProgrammingError   Int String
            | SQLNotSupportedError
```

The `Int` and `String` parts of the error constructors are a database dependent error code and a human readable error message.

The following table describes the meaning of the different constructors.

Error	**Description**
Warning	Non fatal errors, you can still continue
InterfaceError	Error related to the interface, not the database itself
DatabaseError	Error related to the database that can not be classified as Operational error or Internal error
DataError	Error due to problems with the data
OperationalError	Error due to operational problems with the database. E.g. disconnects, memory full etc.
IntegrityError	Errors related to data integrity, e.g. key constraint violations
InternalError	Errors related to internal problems in the database library
ProgrammingError	Errors of the end user, e.g. syntax errors in SQL statements
NotSupportedError	An operation is not supported by the database library

Environments, Contexts and Connections

Communication with a database usually requires some initialization. In the Clean SQL API, this is done in three phases. First you initialize a database library in an `SQLEnvironment`, this will give you an `SQLContext` in which you can open an `SQLConnection`. You can open multiple connections in a single context, so you have to initialize the library only once. Finally you can open an `SQLCursor` on a connection. This cursor can be used as a handle which allows you to interact with a database.

Since the API abstracts over different database libraries, the `SQLEnvironment`, `SQLContext`, `SQLConnection` and `SQLCursor` are defined as type classes.

```
class SQLEnvironment env ctx
where
    sql_init            :: !*env      → (!(Maybe SQLError), !(Maybe *ctx), !*env)
    sql_end             :: !*ctx !*env → (!(Maybe SQLError), !*env)

class SQLContext ctx con
where
    sql_openConnection  :: !String !String !String !String !*ctx
                                        → (!(Maybe SQLError), !(Maybe *con),!*ctx)
    sql_closeConnection :: !*con !*ctx  → (!(Maybe SQLError), !*ctx)

class SQLConnection con cur
where
    sql_openCursor      :: !*con        → (!(Maybe SQLError), !(Maybe *cur), !*con)
    sql_closeCursor     :: !*cur !*con  → (!(Maybe SQLError), !*con)
```

Except for the `sql_openConnection` function, the arguments of the functions in these type classes should be self explanatory. The first four `String` arguments of `sql_openConnection` are respectively: the host name of the database server, a user name, a password and the name of the database to use.

Cursors

Database cursors are used to interact with the database. The functions that can be used with a cursor handled are defined in the `SQLCursor` type class.

```
class SQLCursor cur
where
    sql_execute         :: !SQLStatement ![SQLValue] !*cur
                                         → (!(Maybe SQLError), !*cur)
    sql_executeMany     :: !SQLStatement ![[SQLValue]] !*cur
                                         → (!(Maybe SQLError), !*cur)
    sql_numRows         :: !*cur         → (!(Maybe SQLError), !Int, !*cur)
    sql_numFields       :: !*cur         → (!(Maybe SQLError), !Int, !*cur)
    sql_insertId        :: !*cur         → (!(Maybe SQLError), !Int, !*cur)
```

```
sql_fetchOne      :: !*cur       → (!(Maybe SQLError), !(Maybe SQLRow), !*cur)
sql_fetchMany     :: !Int !*cur  → (!(Maybe SQLError), ![SQLRow], !*cur)
sql_fetchAll      :: !*cur       → (!(Maybe SQLError), ![SQLRow], !*cur)
```

The arguments and return values of these functions are not immediately clear, so the following table explains them each individually.

Function	**Description**
sql_execute	This function executes an SQL statement. In this statement "?" markers may be placed which are replaced by the values from the second argument. The length of the list of values must be equal to the number of "?" markers in the SQL statement. If not, a programming error is returned.
sql_executeMany	This function does the same as the previous, but multiple times. Therefore a list of lists of values is given. Depending on the possibilities of the database, this can be implemented as executing a single query multiple times, or optimized by sending the query once, and just the values multiple times.
sql_numRows	This function has a different meaning depending on the type of statement that was sent last. If it was a `SELECT` statement, the number of found rows is returned. If it was an `INSERT` statement it returns the number of inserted rows. When an `UPDATE` or `DELETE` statement was executed, the returned value is the number of rows matching the `WHERE` selection part of the statement.
sql_numFields	This function returns the number of fields that the result set rows will have. In other words, the length of the result rows.
sql_insertId	When a record is inserted in a table with an auto-incrementing primary key, this function returns the last inserted number.
sql_fetchOne	This function returns the next result row after executing a statement, or `Nothing` if there are no result rows left.
sql_fetchMany	This function returns *at most* the next `n` result rows. If there are less then `n` left, everything that could be fetched is returned.
sql_fetchAll	This functions returns all the result rows after executing a statement.

Database Layout

The Clean Database API does not only specify types and functions for interacting with an existing database, it also defines types for representing the structure of databases themselves.

```
:: SQLDatabaseName   :== String
:: SQLTableName      :== String
:: SQLColumnName     :== String

:: SQLType          =   SQLTChar Int
                    |   SQLTVarchar Int
                    |   SQLTText
                    |   SQLTInteger
                    |   SQLTReal
                    |   SQLTFloat
                    |   SQLTDouble
                    |   SQLTDate
                    |   SQLTTime
                    |   SQLTTimestamp
                    |   SQLTDatetime
                    |   SQLTEnum [String]
                    |   SQLTUnknown Int

:: SQLTable         = { name            :: SQLTableName
                    ,   columns         :: [SQLColumn]
                    ,   primary_key     :: SQLPrimaryKey
                    ,   foreign_keys    :: [SQLForeignKey]
                    }

:: SQLColumn        = { name        ::  SQLColumnName
                    ,   type        ::  SQLType
                    ,   null        ::  Bool
                    ,   default     ::  Maybe SQLValue
                    }

:: SQLPrimaryKey     :== [SQLColumnName]

:: SQLForeignKey    = { columns         ::  [SQLColumnName]
                    ,   ref_table       ::  SQLTableName
                    ,   ref_columns     ::  [SQLColumnName]
                    ,   update_action   ::  SQLRefAction
                    ,   delete_action   ::  SQLRefAction
                    }

:: SQLRefAction     = SQLCascade
                    | SQLRestrict
                    | SQLNoAction
                    | SQLSetNull
                    | SQLSetDefault
```

```
class SQLTableInfo con
where
    sql_tables          :: !*con      → (!(Maybe SQLError), ![SQLTableName], !*con)
    sql_describe        :: !SQLTableName !*con
                                      → (!(Maybe SQLError), !(Maybe SQLTable), !*con)
```

If a database library provides an instance of the `SQLTableInfo` class for its connection type, the structure of a database can be extracted.

Utility functions

Finally the Clean SQL API module gives instances for equality and string formatting for the commonly used types.

```
instance toString SQLType
instance toString SQLValue
instance toString SQLError

instance == SQLType
instance == SQLValue
```

Appendix B

The GenSQL library

Basic Types

The generic `gSQL` works on a token list, and maintains a field type information list. Since it also is actually six functions at once, there are also types to instruct the function what to do. In normal usage of the GenSQL library, you never need to deal with these types. They are exported only for the rare cases that the generic function has to be specialized for a non standard scalar.

```
:: GSQLMode      = GSQLCreate
                 | GSQLRead
                 | GSQLUpdate
                 | GSQLDelete
                 | GSQLInfo
                 | GSQLInit

:: GSQLPass :== Int

:: GSQLToken     = GSQLValue SQLValue
                 | GSQLTerminator
                 | GSQLOverride String SQLValue

:: GSQLFieldInfo =  { fld_table      :: String
                    , fld_select     :: Maybe String
                    , fld_match      :: Maybe String
                    , rec_table      :: String
                    , rec_key        :: String
                    , val_list       :: Bool
                    , val_maybe      :: Bool
                    , val_fields     :: [GSQLFieldInfo]
                    , val_id         :: Bool
                    }
```

Errors

In working with the mapping, two things can go wrong. The first, most likely problem, is that a problem in the database. In this case, the error the database API returned is simply passed along. Another thing that can be wrong, is misuse of the mapping by the programmer. Because the generic function is by definition defined for every type, it can be applied to values which make no sense in a database context. In other words, these types are not part of the set of representation types. In these cases a run time type error is given.

```
:: GSQLError    = GSQLDatabaseError SQLError
                | GSQLTypeError String
```

The Generic function

All the real mapping work is performed by the `gSQL` generic function. It has already been explained in detail in chapter 4 and needs no further explanation.

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

Operation Wrappers

The generic mapping is used by means of the operation wrapper functions. There use has also been explained in chapter 4, and demonstrated in chapter 6.

```
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
```

Utility functions

Finally the GenSQL library also defines one utility function. Namely the instance of `toString` for the `GSQLError` type.

```
instance toString GSQLError
```

Bibliography

[1] Sinam Si Alhir, *UML in a nutshell*, O'Reilly, Sep 1998.

[2] Artem Alimarine and Rinus Plasmeijer, *A Generic Programming Extension for Clean*, The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers (Thomas Arts and Markus Mohnen, eds.), LNCS, vol. 2312, Springer Verlag, Sep 2002, pp. 168–186.

[3] 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.

[4] Björn Bingert and Anders Höckersten, *Student paper: Haskelldb improved*, Proceedings of 2004 ACM SIGPLAN workshop on Haskell, ACM Press, 2004, pp. 108–115.

[5] Gilad Bracha, *Generics in the java programming language*, http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, 2004.

[6] Paul DuBois, *The MySQL C API*, MySQL, New Riders, 1999, pp. 221–273.

[7] Mark Fussel, *Foundations of object-relational mapping*, http://www.chimu.com/publications/objectRelational/index.html, 1997, Whitepaper.

[8] Terry Halpin, *Information modeling and relational database: from conceptual analysis to logical design*, Morgan Kaufmann Publishers Inc, 2001.

[9] *The Hibernate Homepage*, http://www.hibernate.org/.

[10] Ralf Hinze, Johan Jeuring, and Andres Löh, *Comparing approaches to generic programming in haskell*, Lecture notes of the Spring School on Datatype-Generic Programming 2006 (Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, eds.), LNCS, vol. 4719, Springer Verlag, 2007, pp. 72–149.

[11] 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.

[12] Marc-André Lemburg, *Python database API specification v2.0*, http://www.python.org/dev/peps/pep-0249/, 2008.

[13] 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.

[14] *The MySQL Homepage*, http://www.mysql.com/.

[15] Ulf Norell, *Dependently typed programming in agda*, Tech. Report ICIS-R08008, Radboud University Nijmegen, 2008.

[16] Betsy Pepels and Rinus Plasmeijer, *Generating applications from object role models*, On the Move to Meaningful Internet Systems 2005: OTM Workshops (R Meersman et al., eds.), LNCS, vol. 3762, Springer Verlag, 2005, pp. 656–665.

[17] Rinus Plasmeijer and Peter Achten, *iData For The World Wide Web - Programming Interconnected Web Forms*, Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006), LNCS, vol. 3945, 2006.

[18] Rinus Plasmeijer, Peter Achten, and Pieter Koopman, *iTasks: Executable Specifications of Interactive Work Flow Systems for the Web*, Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007) (Freiburg, Germany), ACM, Oct 1–3 2007, pp. 141–152.

[19] Rinus Plasmeijer and Marko van Eekelen, *Concurrent CLEAN language report (version 2.0)*, Tech. report, University of Nijmegen, Dec 2001.

[20] Jean-Jacques Wintraecken, *Informatie-analyse volgens NIAM: in theorie en praktijk*, Schoonhoven: Academic Service, 1987.