Building a Functional Relational Database

J.C. ter Braak
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.c.terbraak@student.utwente.nl

W.R.C. Drijfhout
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
w.r.c.drijfhout@student.utwente.nl

ABSTRACT

We present a proof-of-concept for a functional database implementation which implements the Relational Data Model more accurately than conventional databases, and offers a more elegant solution to storing arbitrary data types, including functions, as *first-class citizens.*
Additionally, we propose a composable architecture for data containment and database features, and an interface for implementing these, along with some example implementations. We show how this concept is feasible and useful, and demonstrate some examples of how these can be used effectively in real-world situations.

Keywords

Database, Database Management System, Functional, Relational, Data model, Haskell

1. INTRODUCTION

Databases are used to contain data about some state of affairs in a world model, by mapping and saving it to a certain data model. One highly influential data model is Codd's Relational Data Model[3]. It maps real-world data values to *domains*, and defines a *relation* as a mapping between values, one from each domain. These *relations* can be seen as non-injective, non-surjective functions, that map primary keys to elements from the domains defined in the relation. However, they are not the same as functions, but rather a subset, due to the constraint that they must map to each of the domains in the relation. The relational data model was a revolutionary approach in perspective to existing popular data models such as the hierarchical model and the network model. However, while the industry adopted the relational data model, it made compromises.

Where Codd did not prescribe specific data types to use as domains, database manufacturers restricted the support for data types to a predefined set. Support for composite data types was added later on, but the distinction between primitive and composite types still remains. As commercial databases are optimized for performance rather than versatility or elegance, the underlying data structures only contain primitive data types, and additional functionality is implemented as separate vendor-specific modules built on top of the database engine. As we conclude from [4], all extra functionality that is normally implemented separately from the data storage, could be implemented as functions in the data itself, thereby evaluating them as the data is accessed, and only activating functionality when it is actually used.

Using the functional paradigm in databases could prove very useful here: for both Database Management System (DBMS) functionality as well as for implementing the relational data model. Characteristics such as parameter currying, lambda abstractions and generalization of functions and values as data types allow us to store functions into our database, and to modify the implementation of various parts of the database at runtime. Various proposals for a functional database already exist, but these are merely conventional databases accessed with a functional query language. We propose a true *functional* database, the functional relational database.

As one might ask how such a functional relational database would work in practice, we will establish requirements and propose an architecture which will form the basis for a functional relational database.

2. REQUIREMENTS

The requirements to a functional relational database can be derived from the purpose of functionality in existing database solutions.

One of the key characteristics of modern databases is their compliance with Codd's relational data model[3], which allows for easy modeling of real world data as relations, implemented as mappings between domains. We will need to implement such functionality in a functional relational database as well. The relational data model does not prescribe what data types to use as elements[10], but nonetheless most conventional databases implement a set of primitive data types, next to functionality for defining composite types. An improvement would be to implement functionality that does not discriminate between primitives and composite types, by eliminating the distinction altogether. One important consequence would be the ability to store functions as *first-class citizens* (some constraints apply to a relation's primary key; discussed in 4.4.1). Some conventional databases have support for stored procedures, but these are vendor-specific and need to be created and evaluated explicitly. If we could store functions implicitly, the DBMS could be agnostic of whether a data item is a function or a literal value, as long as they return the same data type. This would offer a great degree of flexibility for both the database administrator and the user, as they are not forced to use either functions or literal values for certain data.

15^{th} *Twente Student Conference on IT* June 20^{th}, 2011, Enschede, The Netherlands

Another requirement to a database is a comprehensive authorization mechanism. To illustrate this, we will create the scenario of a car dealership, where new and used cars are being administrated in a database. Various employees should ideally be given different privileges. For example, a technician should typically be allowed to update a car's service history, but not delete it, or change the car's price tag. On the other hand, a sales representative should be allowed to change the price tag, but only on the cars that he is currently allowed to sell. A per-attribute and per-element authorization policy would be required to implement this, but in conventional databases this is generally seen as too impractical. Therefore the authorization policy is usually only implemented in the relation and database scopes. To support more finely-grained authorization policies, the DBMS allows the user to create custom views, artificial relations that are composed of transformations of existing relations. This approach is effective but usually cumbersome to set up and maintain. We would like to define aspects like versioning, authorization, and in a more general sense, context dependency in all scopes, thereby embedding such detailed information into the data itself. Additionally, the constraints put on data should also apply on its parent levels. For example, if a data element employs a certain authorization policy, then the same policy should be enforced when the data element is encapsulated into a domain, or into a relation. This is similar to the principle of *level shifting* in [1], and has been named *lifting* by [4]. For the purpose of this document, we will adopt the latter naming.

One of the most important aspects of a database is the ability to query for data. An eventual implementation will have to specify a relationally complete[3] query language in order to access the data. It is important to note that for functions with matching type signatures it can fundamentally not be decided whether or not they are the same function. Therefore a compromise will have to be made in implementing functions as *first class citizens*, to allow them to be equated. One solution would be to require the user to evaluate the compared functions during querying, and define them to be equal if they return the same result for a particular set of arguments. Querying and query optimization, however, out of the scope of this research, and would be a subject for future work.

Given these properties, we can establish the most important requirements to be the following:

1. A functional relational database must allow functions to be defined as first-class citizens.
2. A functional relational database must implement Codd's relational data model.
3. A functional relational database must be able to create, read, update, and delete arbitrary data types.
4. A functional relational database must allow data to be defined context-dependently on each level. These context-dependencies should be *lifted* to their parent levels.

3. RELATED WORK

One existing approach to using the functional paradigm in databases is HaskellDB[2, 8]. Rather than being a functional relational database implementation, HaskellDB is a functional library for accessing traditional relational databases. Therefore it is not so much an existing solution for our problem, as it is a framework that offers a functional perspective on databases. It presents database records as types and the classic SQL operations as functions that project, restrict and manipulate data. Databases that use a similar query language are frequently named Functional Databases, however they add nothing to the concept of the Relational Database except for a functional programming method of access.

[6] proposes a simplistic but elegant functional implementation of a relational database, along with a basic data manipulation interface. The implementation language being used, namely Haskell, represents data references as type arguments in constructors. Implementing circular references is impossible using this scheme, therefore this design implements foreign keys using database reference surrogates, which are essentially proxies for the actual data records. While this implementation accepts arbitrary data types, it offers no facilities for saving and evaluating their context-dependency, and therefore does not meet our Requirement #4.

Another approach to databases from a functional perspective is [11]: it shows an effective way of mapping real-world domains to a database, by modeling data items as algebraic data types, and relations between them as functions. However, the author does not present a real database access method or a way of defining data context-dependently. Therefore this approach is insufficient for implementing a functional relational database, as it does not satisfy Requirement #3 and #4.

4. ARCHITECTURE

To implement a functional relational database, we would need an architecture that allows functions to be stored as regular data types. One option would be to use an existing DBMS and extend its functionality to include functions as *first-class citizens*. Many databases have support for *Binary Large Objects* (BLOBs) or *Abstract Data Types* (ADTs), which could be used to store functions as byte code as proposed in[9], or as stored procedures. We would need a separate interpreter to execute these functions. The greatest advantage is that we wouldn't have to "reinvent the wheel", but we would be tied to the limitations of existing functionality. Instead, it would be more practical to create a completely new implementation, which would not leave us restricted to using the existing functionality. As functional language for the implementation, Haskell[5] has been used.

4.1 Context

In [4], Drijfhout has presented various context aspects which could be relevant during data retrieval and manipulation:

- Relational Data Model: Values in a domain relate to their primary key
- Authorization: The ability to access data depends on the privileges of the user accessing the database
- Versioning: The data we store or retrieve may be different depending on the referenced time frame

Using these aspects, we can define a *Context*-type which we will pass to all our database operations:

```
data Context pkType = Context { pk   :: pkType,
                                user :: String,
                                time :: Int }
```

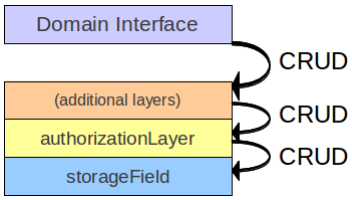

Figure 1. Layered data access model

It is possible to use any data type that is an instance of the *Eq*-class as primary key (In Object Oriented Programming terms, this can be thought of as implementing the Equatable-interface). For every data operation, a *Context* will be constructed by the DBMS which includes all relevant context information. An example instance of this *Context* type could look like the following:

```
Context {
 pk   = 1,
 user = "John",
 time = 1304699109
}
```

Every database operation accepts a *Context* and may use the data within in its execution. Not every operation requires all *Context* fields to be set to meaningful values, and not every user-initiated operation is able to set every field to a meaningful value. For example, in a query where a user retrieves a complete relation, the primary key field is irrelevant and not used by the underlying operation, and may be set to an arbitrary value.

4.2 A Generic Way of Containing Data

Composability. In [4], Drijfhout has proposed a way of implementing the *create*, *read*, *update* and *delete* (CRUD)-functions explicitly inside the database, and having the DBMS pass a Context-type when calling these functions. For each distinct type of domain we wish to store, such as traditional key-value data, continuous functions or composite types, we will have to implement these functions accordingly. This CRUD-interface not only allows us to create various ways of storing data, but also forms the foundation for a composable data access model. As shown in the layered structure in Figure 4.2, the interface that accesses the data evaluates the CRUD-functions of the top *data access layer*. Each layer then calls the corresponding function of the underlying layer, and applies its own logic to it (this can be thought of as the *decorator*-pattern in Object Oriented Programming). For example, an authorization layer will enforce the authorization policy for the field before allowing calls to underlying layers to pass through. The CRUD-function calls propagate down the layered structure, and eventually reach the storageField-layer (which we will discuss later on), which is the *container* that actually stores the values.
This approach allows us to create a library of data access layer default implementations, out of which various data containers can easily be composed. For example, if a user wishes to create a discrete finite domain that employs authorization, versioning and data validation, we just pick a suitable container implementation from the library, and stack the corresponding data access layers (also from the same library) on top of it. If some data access layer with a custom implementation is required, it can easily be created, as long as it adheres to the specification.

Container. Below is an overview of the the *Container* type which every implementation of a data container or data access layer will implement:

```
data Container pkType dataType =
  Container {
    fcreate   :: Context pkType
              -> Container pkType dataType
              -> ...

    fread     :: Context pkType
              -> Container pkType dataType
              -> ...

    fupdate   :: Context pkType
              -> Container pkType dataType
              -> ...

    fdelete   :: Context pkType
              -> Container pkType dataType
              -> ...
  }
```

The primay key type and the stored data type are parametrized in the *Container* type, and will be inferred automatically depending on how we use them. As we can see, each of the CRUD-functions takes an instance of the *Context*-type. The implementation of each of these functions may choose whether or not to make use of this information while manipulating the data. As a second argument, the current database state is being passed to the functions, from which they will derive the new database state. Below, we will explain the slightly different type signature of each of the functions.

Create. The *create*-function takes an instance of *dataType* (the value we wish to save), and returns an updated *Container*, which represents the new state of the database. *Create* requires that no value already exists.

```
fcreate   :: Context pkType
          -> Container pkType dataType
          -> dataType
          -> IO (Container pkType dataType)
```

Note: The *IO*-type is a special data type that wraps the result inside the *IO*-monad. The *IO*-monad is a Haskell-specific way of handling input and output; it separates the "pure" part of the program from the "tainted" part by wrapping all actions that involve I/O operations. This allows us to print status and error messages during operations, while still returning a consistent database state.

Read. The *read*-function just returns the value present in this *Container*, or *Nil* of no such value exists. Contrary to the other CRUD-functions, *read* does not create a new database state.

```
fread     :: Context pkType
          -> Container pkType dataType
          -> IO (Maybe dataType)
```

Note: The *Maybe*-type is a special data type that allows is to either return the result, or a *Nothing*-value if retrieving the result failed (for example, if it didn't exist).

Update. The *update*-function works similar to the *create*-function but requires an existing value to already be present in the *Container*. Like *create*, it returns the new database state.

```
fupdate   :: Context pkType
          -> Container pkType dataType
          -> dataType
          -> IO (Container pkType dataType)
```

Delete. The *delete*-function returns an updated *Container*, representing the new database state with the value erased:

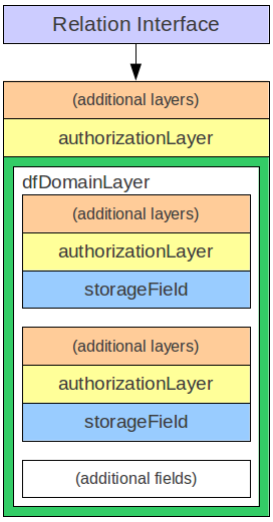

Figure 2. Domain Structure

```
fdelete    :: Context pkType
           -> Container pkType dataType
           -> IO (Container pkType dataType)
```

As we can see, the *create*, *update* and *delete*-functions each return an updated database state. However, if the operation fails, they will print an error message and return the current, unaltered database state.
As each of the CRUD-layers uses the same functions and *Context* to determine how it should read or manipulate the data, it is a relatively simple task to *lift* this functionality to the domain level. If we wish to use a *Container* as a domain, it will need to save multiple values which depend on their primary key. The primary key will be supplied in the *Context*, and the implementation of the CRUD-functions will store and load values accordingly. This gives us control on the domain level. However, we still wish to maintain our control and context-dependency on the field level as well. To achieve this, the domain layer will simply be a *Container* which takes a *Container* as dataType, as illustrated in Figure 4.2. Each data element will reside in an individual *Container*, and these *Containers* are themselves being saved inside a domain-level *Container*. This architecture creates endless possibilities for creating complex data collections, but for the purpose of this research we will stick to the traditional relational data model which employs elements, domains and relations.

4.3 An Actual Implementation

To illustrate the precise workings of the generic *Container* data type we will show a specific implementation of it, namely *storageField*, which stores a single value. As previously stated, it implements the CRUD-functions explicitly, and this defines how this *Container* stores and retrieves data. Note that this is just one of many possible implementations; it is perfectly possible to make a different implementation for each type of data containment we need.

The most important aspect of every implementation of Container revolves around modifying the contents of the *read*-function. This is due to the fact that the *read*-function defines what values are currently present in the *Container*, and how they are read.

Container. The general structure of the *Container* is analogous to its type definition:

```
storageField =
 Container {
  fcreate
    = \c
   -> \this
   -> ...,

  fread
    = \c
   -> \this
   -> ...,

  fupdate
    = \c
   -> \this
   -> ...,

  fdelete
    = \c
   -> \this
   -> ...
 }
```

Create. Firstly, we have defined the *create*-function (in this case called *fcreate*). It takes a *Context*, the current database state and a value. It checks whether a value already exists using the read function on the current database state (`result <- fread this c this`; note that this function call takes the *this* argument twice. This is a consequence of Haskell's record design: `fread this` is a reference to the *read*-function inside *this*, the *Container*. This function then requires a *Context* and a *Container* to perform its operation, hence the second reference to *this*). If the value exists, *create* prints an error message and returns the current, unaltered database state. Otherwise it creates a new *Container*, representing the new database state. When we later evaluate the *read*-function in the new database state, it will return the stored value:

```
fcreate
  = \c -> \this -> \v ->
    do
      result <- fread this c this
      case result of
        Just _ ->
        do
          putStrLn "Trying to create an already
              existing value."
          return this
        _       ->
        return Container {
          fcreate   = fcreate this,
          fread     = \c_ -> \this_ -> return (
              Just v),
          fupdate   = fupdate this,
          fdelete   = fdelete this
        }
```

Note: The new *read*-function contains a completely new lambda abstraction which is independent from one in the *create*-function. As its lambdas are different from the ones in *create*, they have been marked with an underscore.

Read. Next, we defined the *read*-function which is initially very simple. As a newly created *Container* should not contain any values, the *read*-function returns a *Nothing*, indicating that retrieving the value was unsuccesful.

```
fread
  = \c
 -> \this
 -> return Nothing
```

Update. The third function in the *Container*, the *update*-function is similar to the *create*-function, except for the fact that it requires a previous value to be present (Note the substitution of *Nothing* for *Just _*):

```
fupdate
  = \c -> \this -> \v ->
```

```
    do
      result <- fread this c this
      case result of
        Nothing ->
        do
          putStrLn "Trying to update a non-
              existing value."
          return this
        _       ->
        return Container {
          fcreate   = fcreate this,
          fread     = \c_ -> \this_ -> return (
              Just v),
          fupdate   = fupdate this,
          fdelete   = fdelete this
        }
```

Delete. Finally we include the *delete*-function, which is the same as the *update*-function, but resets the *read*-function to return *Nothing*.

```
fdelete
  = \c -> \this ->
    do
      result <- fread this c this
      case result of
        Nothing ->
        do
          putStrLn "Trying to delete a non-
              existing value."
          return this
        _       ->
        return Container {
          fcreate   = fcreate this,
          fread     = \c_ -> \this_ -> return
              Nothing,
          fupdate   = fupdate this,
          fdelete   = fdelete this
        }
```

Using this storage field implementation we can store and retrieve a single arbitrary data type. However, it does not contain any functionality for handling context dependency.

4.4 Introducing Context Dependency

The implementation in the previous paragraph allows us to store simple values, independent of any context. However, if we want to store an entire domain, we will have to map multiple values and how they relate to their respective primary keys in the relation. This introduces the first context dependency: values depend on their primary key.

4.4.1 Primary Key

Domain. A domain definition is required to save multiple values in the same domain of a relation. It introduces a *subLayer* parameter which denotes the underlying type of field that should be used to store data. This type is then used in the contained CRUD-functions:

```
dfDomainLayer subLayer = Container {
  fcreate = ...,
  fread   = ...,
  fupdate = ...,
  fdelete = ...
}
```

Note the following implementation of the CRUD-functions that store values dependent on their primary key. It simply adds an extra layer on top of the underlying *subLayer*: Initially the *read*, *update* and *delete*-functions don't perform any operations, since the domain is initally empty. Once a value is created using the *create*-function (left a stub for now), the new database state will contain variants of *read*, *update* and *delete* that include the newly created value, and check for a matching primary key before executing their operations:

```
dfDomainLayer subLayer = Container {
  fcreate
    = \c -> \this -> \v -> ...,

  fread
    = \c -> \this -> return Nothing,

  fupdate
    = \c -> \this -> \v ->
      do
        putStrLn "Trying to update a non-
            existing value."
        return this,

  fdelete
    = \c -> \this ->
      do
        putStrLn "Trying to delete a non-
            existing value."
        return this
}
```

Create. The implementation of the create-function is somewhat complex, and therefore broken down below. Firstly, *create* checks whether a value already exists using the *read*-function. If so, it prints an error and returns the current, unaltered database state:

```
fcreate   = \c -> \this -> \v ->
 do
  result <- fread this c this
  case result of
   Just _ ->
   do
     putStrLn "Trying create an already existing
         value."
     return this
  _       -> ...
```

Otherwise, it creates the value in the underlying *Container*:

```
    do
     createdValue <-fcreate subLayer c subLayer v
```

New Database State. After the value has been created, the function returns the new database state.

```
    return Container {
      fcreate   = ...,
      fread     = ...,
      fupdate   = ...,
      fdelete   = ...
    }
```

Again, the CRUD-functions have been stubbed for readability. Each of the functions is described below.
The new *create*-function simply refers to the domain's *create*-function as it requires no different implementation in the new database state:

```
      fcreate   = fcreate this
```

The new *read*-function returns the value we just created, but only if the primary key used to *create* equals the one being used to *read*. Otherwise, it executes the already existing *read*-function, which contains the previously created values.

```
      fread     = \c_ -> \this_ ->
        if pk c == pk c_ then
          fread createdValue c_ createdValue
        else fread this c_ this
```

If the primary key that has been used to *create* matches the one being used to *update*, the new *update*-function updates *createdValue*, and returns a new database state with the updated value added to the *read*-function. If the primary key doesn't match, it passes to call to the already existing *update*-function, which contains the previously created values.

```
    fupdate   = \c_ -> \this_ -> \v_ ->
      if pk c == pk c_ then
      do
        updatedField <- fupdate createdValue
            c_ createdValue v_

        return Container {
          fcreate   = fcreate this_,
          fread     = \c__ -> \this__ ->
            if pk c_ == pk c__ then
              fread updatedValue c__
                  updatedValue
            else fread this_ c__ this_,
          fupdate   = fupdate this_,
          fdelete   = fdelete this_
        }

      else fupdate this c_ this v_,
```

The *delete*-function deletes the value from the underlying *Container* if the primary key matches, otherwise it executes the already existing *delete*-function.

```
    fdelete   = \c_ -> \this_ ->
      if pk c == pk c_ then
       fdelete createdValue c_ createdValue
      else fdelete this c_ this
```

4.4.2 Authorization

To implement authorization into our database, we add the *authorizationLayer* to our architecture, analogous to *dfDomainLayer*. Its structure is similar to *dfDomainLayer*, but instead of checking for a primary key, it checks whether the user executing the operation is allowed to do so. The following is a default implementation for *authorizationLayer* that allows any user to create or read values, but values may only be updated or deleted by their creator. If a different policy is desired, a separate *authorizationLayer* implementation will be required.

```
authorizationLayer subLayer = Container {
  fcreate   = \c -> \this -> \v ->
  do
    result <- fread this c this
    case result of
      Just _ ->
      do
        putStrLn "Trying to create an already
            existing value."
        return this
      _       ->
      do
        newSubLayer <- fcreate subLayer c
            subLayer v

        return Container {
          fcreate   = fcreate this,

          fread     = fread newSubLayer,

          fupdate   = \c_ -> \this_ -> \v_ ->
            if user c == user c_ then
            do
             updatedSubLayer <- fupdate
                 newSubLayer c_ newSubLayer v_

             return Container {
              fcreate   = fcreate this_,
              fread     = \c__ -> \this__ ->
                if user c_ == user c__ then
                 fread updatedSubLayer c__
                     updatedSubLayer
                else fread this_ c__ this_,
              fupdate   = \c__ -> \this__ -> \
                  v__ ->
                if user c_ == user c__ then
                 fupdate updatedSubLayer c__
                     updatedSubLayer v__
                else fupdate this_ c__ this_ v__
                    ,
              fdelete   = \c__ -> \this__ ->
                if user c_ == user c__ then
                 fdelete updatedSubLayer c__
                     updatedSubLayer
                else fdelete this_ c__ this_
             }
            else
            do
             putStrLn ("Access denied to user "
                 ++ user c_ ++ ".")
             return this_,

          fdelete   = \c_ -> \this_ ->
            if user c == user c_ then
             fdelete newSubLayer c_ newSubLayer
            else
            do
             putStrLn ("Access denied to user "
                 ++ user c_ ++ ".")
             return this_
        },

    fread     = ..., [see dfDomainLayer]
    fupdate   = ..., " "
    fdelete   = ...  " "
  }
```

4.4.3 Additional Layers

Taking the *authorizationLayer* as an example, we could implement various other data access layers that implement additional functionality like versioning and offer elegant solutions to problems like schema evolution, by implementing a transformation function. The architecture allows us to stack an arbitrary amount of layers, in an arbitrary order, and thereby offers a very flexible framework for defining relations.

4.5 Relations

An important characteristic of the functional relational database is the ability to define relations as mappings between values from various domains. Theoretically, our implementation of relations is fairly straightforward. As shown in Figure 4.5, we simply create a data access layer that decomposes records into multiple domains when they are stored, and composes them again on retrieval. However, in a functional language like Haskell this is no trivial task as our domains may be different types depending on the type of values they store. Haskell has no suitable and simple support for heterogeneous collections. Initiatives like HList[7] and HDBRec[8] exist, however they are fairly complex and therefore implementing a generic *relationLayer* is a daunting undertaking. For the sake of this proof-of-concept we have decided to implement the *relationLayer* separately for each relation we would like to store, as we can then use records instead of heterogeneous collections.

4.5.1 Example Relation: Products

Consider the case where we would like to store products of the following form:

```
data Product = Product {
                product_Name  :: IO String,
                product_Price :: IO Float
               }
```

As we cannot generate a set of domains from this record declaration due to the limitations of Haskell, we will need to specify the domains to use in a dedicated record:

```
data ProductDomains =
  ProductDomains {
    product_NameDomain  :: IO (Container String)
        ,
    product_PriceDomain :: IO (Container Float)
  }

productDomains =
  ProductDomains {
    product_NameDomain  =
      return (dfDomainLayer (authorizationLayer
          (storageField))),
    product_PriceDomain =
```

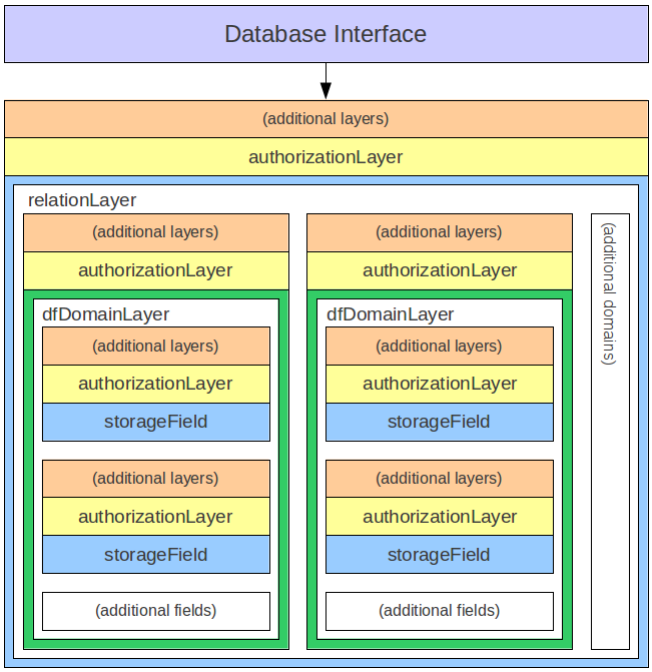

Figure 3. Relation Structure

```
    return (dfDomainLayer (authorizationLayer
        (storageField)))
}
```

Our implementation of the *products* relation will then use these domains including their access layers to store records. The fact that we cannot simply pass a list of domains to the *relationLayer* complicates things, and therefore the *products* implementation deviates slightly from the previous data access layer implementations. The products *Container* takes the *productDomains* as subLayer. The *create*-function in the products *Container* uses the *productDomains* to find the matching domain to every attribute of the Product we wish to store, and executes the corresponding operation on that domain:

```
fcreate   = \c -> \this -> \v ->
do
  result <- fread this c this
  case result of
    Just _ ->
    do
      putStrLn "Trying to create an already
          existing value."
      return this
    _      ->
    return (products productDomains {
      NameDomain =
      do
        container <- NameDomain subLayer
        value <- product_Name v
        fcreate container c container value,
      PriceDomain =
      do
        container <- PriceDomain subLayer
        value <- product_Price v
        fcreate container c container value
    }),
```

The remaining CRUD-functions work the same way by passing the call to each of the domains in the record. The downside of this approach is that the Relation-layer will have to be reimplemented for every relation we create, but as this implementation is straightforward and similar for every relation, this code can easily be generated by the DBMS or a separate utility.

5. TESTING

To test our implementation, we will construct various relations and database schemata. The testing will be done through GHCi, the interactive Haskell interpreter.

5.1 Requirement #1: Functions as First-Class Citizens

Our database architecture allows for the storage of arbitrary data types, including lambda abstractions. We have defined a BankAccount datastructure, and the relation to go along with it:

```
data BankAccount =
  BankAccount {
    accountNumber :: IO Int,
    balance       :: IO Double,
    interestFunc  :: IO (Double -> Int -> Double
        )
  }
```

The *interestFunc*-function takes a *balance* and a time interval in seconds, and calculates a new balance. The following example demonstrates its usage. We have defined a bank account with a balance of €10.00, and an interest of $2 * 10^{-7}$percent per second. We can now save this bank account to the database:

```
*Main> let accounts = bankAccounts
    bankAccountDomains

*Main> let someBankAccount =
  BankAccount {
    accountNumber = return 1,
    balance = return 10.00,
    interestFunc = return (\bal -> \dt -> bal *
        1.000000002^dt)
  }

*Main> accounts <- fcreate accounts
  Context {
    pk=1,
    user="Me",
    time=0
  }
  accounts someBankAccount
```

Using the *read*-function on the *accounts* database state we can retrieve it again. We can then simply calculate the new balance after one year by applying the retrieved interest function to the retrieved balance and some time interval (namely the equivalent of 1 year in seconds), and notice we have received 65 cents in interest:

```
*Main> account <- fread accounts
  Context {
    pk=1,
    user="Me",
    time=1
  }
  accounts

*Main> interestFunc <- interestFunc (fromJust
    account)
*Main> balance <- balance (fromJust account)

*Main> interestFunc balance 31536000
10.651035214671602
```

5.2 Requirement #2: Relational Data Model

As displayed in 5.1, the database architecture allows for the creation of relations. However, as of yet it does not support primary key-foreign key constructs, or most of relational algebra.

5.3 Requirement #3: Arbitrary Data Types

In 5.1 we have demonstrated how lambda abstractions can be saved into the database just like any other data type. It is also possible to store our own defined algebraic data types:

```
data Tree = Node Tree Tree
          | Leaf

data CompositeTree = CompositeTree {
                       treeName :: IO String,
                       treeRoot :: IO Tree
                     }
```

The CompositeTree we have defined interacts with the database in the same way that other data types do:

```
*Main> let rel0 = compositeTrees
    compositeTreeDomains

*Main> let someTree =
     CompositeTree {
       treeName = return "Example Tree",
       treeRoot = return
                  (Node
                    (Node Leaf Leaf)
                    (Node
                      (Node Leaf Leaf)
                      Leaf
                    )
                  )
     }

*Main> trees <- fcreate rel0
     Context {
       pk="0",
       user="Me",
       time=0
     }
     trees someTree
```

We have now created a new database state *trees*, which contains our *CompositeTree*. If we wish to retrieve it, we execute the *read*-function from the database state:

```
*Main> tree <- fread trees
     Context {
       pk="0",
       user="Me",
       time=1
     }
     trees

*Main> treeName (fromJust tree)
"Example Tree"

*Main> treeRoot (fromJust tree)
Node
  (Node Leaf Leaf)
  (Node
    (Node Leaf Leaf)
    Leaf
  )
```

As we can see, the *read*-function retrieved the stored *CompositeTree*.

5.4 Requirement #4: Context Dependency

Reusing our example from 5.3, we can show that our implementation behaves differently when the Context is changed.

5.4.1 Primary Key

Stored records using the domain layer depend on their primary key. If we save a record with primary key "0", we cannot retrieve it using primary key "1". The database stores and retrieves each record using its own primary key:

```
*Main> let trees = compositeTrees
    compositeTreeDomains

*Main> let someTree =
     CompositeTree {
       treeName = return "Example Tree",
       treeRoot = return
                  (Node
                    (Node Leaf Leaf)
                    (Node
                      (Node Leaf Leaf)
                      Leaf
                    )
                  )
     }

*Main> let anotherTree =
     CompositeTree {
       treeName = return "Just A Leaf",
       treeRoot = return Leaf
     }

*Main> trees <- fcreate trees
     Context {
       pk="0",
       user="Me",
       time=0
     }
     trees someTree

*Main> trees <- fcreate trees
     Context {
       pk="1",
       user="Me",
       time=0
     }
     trees anotherTree
```

We have now created two *CompositeTrees*, called *someTree* and *anotherTree*. These trees have been saved to the database, *trees* being the latest database state. We can now retrieve the trees one by one by modifying our *Context*:

```
*Main> tree <- fread trees
     Context {
       pk="0",
       user="Me",
       time=1
     }
     trees

*Main> treeName (fromJust result)
"Example Tree"

*Main> treeRoot (fromJust result)
Node
  (Node Leaf Leaf)
  (Node
    (Node Leaf Leaf)
    Leaf)


*Main> tree <- fread trees
     Context {
       pk="1",
       user="Me",
       time=2
     }
     trees

*Main> treeName (fromJust result)
"Just A Leaf"

*Main> treeRoot (fromJust result)
Leaf
```

If we pass a *Context* with primary key 0, we get a different result to when we pass a *Context* with primary key 1.

5.4.2 Authorization

Another example is authorization. Using the default implementation of *authorizationLayer*, we can show that values may only be modified by their creator:

```
*Main> let trees = compositeTrees
    compositeTreeDomains

*Main> let someTree =
     CompositeTree {
       treeName = return "Some Tree",
       treeRoot = return Leaf
     }

*Main> let anotherTree =
     CompositeTree {
       treeName = return "Another Tree",
       treeRoot = return Leaf
     }
```

```
*Main> trees <- fcreate trees
      Context {
        pk="0",
        user="Henk",
        time=0
      }
      trees someTree
```

The database state *trees* contains *someTree*, which was created by user "Henk". If we try to update the record with *anotherTree*, but as another user, the operation should fail:

```
*Main> trees <- fupdate trees
      Context {
        pk="0",
        user="Jan",
        time=1
      }
      trees anotherTree

*Main> result <- fread trees
      Context {
        pk="0",
        user="Jan",
        time=2
      }
      trees
Access denied to user Jan.

*Main> treeName (fromJust result)
"Some Tree"
```

An error was returned and the database state still contains *someTree*. However, if we update the record as its creator, "Henk", the operation should succeed:

```
*Main> trees <- fupdate trees
      Context {
        pk="0",
        user="Henk",
        time=3
      }
      trees anotherTree

*Main> result <- fread trees
      Context {
        pk="0",
        user="Henk",
        time=4
      }
      trees

*Main> treeName (fromJust result)
"Another Tree"

*Main> treeRoot (fromJust result)
Leaf
```

6. RESULTS

Implementing a functional relational database is feasible, as demonstrated by this proof-of-concept. We have shown how various DBMS and database functionality can be expressed in a single generic data access layer which implements the CRUD-functions. One of the main advantages of this approach is the fact that data access layers can be stacked and ordered arbitrarily, and the database administrator can pick and choose which layers are required for a particular data container. If additional functionality is required, a new data access layer can easily be implemented and used without having to redesign the entire system.

Not all requirements have been fully met. Implementing the relational data model completely is a significant task, as Haskell does not support heterogeneous collections natively. Workarounds for this limitation exist such as HList[7] and HaskellDB's HDBRec[2], which are fairly complex solutions. As this is a proof-of-concept, implementing one of these would have been beyond the scope of this research.

7. CONCLUSION

Despite the fact that this implementation is far from production ready, it should offer some insight into how a functional relational database works. We have achieved an architecture that works agnosticly from the kind of data types it is storing, which allows us to use functions as *first-class citizens* and store composite types in exactly the same way as primitive data types. Due to this fact, the architecture we propose forms a basis for a more close implementation of Codd's Relational Data Model, compared to commercial solutions.

By forcing each data access layer to implement the CRUD-interface, we ensure that the various data access layer implementations stay interchangable and compatible. The database has a great degree of composability: the behaviour of individual data containers can easily be altered by adding, removing or switching out one of their data access layers. When new functionality is required, or some of it needs to be changed, only the affected data access layer needs to be added or modified, and the rest of the system remains untouched. Each feature can be isolated into its own module, and there is no need to reimplement features for each layer in the database structure.

Combining these facts, we conclude that a database management system can be lightweight, elegant and easily maintainable. The ability to store arbitrary data types and define behaviour in every database layer removes the need for cumbersome features like stored procedures and views. We have created the basics for an architecture that more closely resembles what a relational database is supposed to be.

8. FUTURE WORK

Some work still needs to be done in order to transform this implementation into a production ready database system. Some of the things that could improve this architecture are listed below:

- Implementation of a generic relation layer using heterogeneous collections (for example: HList or HDBRec), to replace the current relation-specific implementations
- A generic database layer which contains a collection of Relations (analogous to the relation layer)
- Support for a relationally complete query language
- More data access layer default implementations for features like versioning, schema evolution and transactions
- A database management system which is more sophisticated than Haskell's GHCi's command line interface

These additions will turn this proof-of-concept into a production ready database system.

ACKNOWLEDGEMENTS

We would like to thank dr.ir. J. Kuper and dr.ir. M. van Keulen for their supervision, reviews, feedback and continuous support of this research project, and C. P. R. Baaij MSc. for his helpful comments and suggestions.

9. REFERENCES

[1] H. Balsters, R. de By, and R. Zicari. Typed Sets as a Basis for Object-Oriented Database Schemas. In *ECOOP '93*, pages 161–184, 1993.

[2] B. Bringert, A. Höckersten, C. Andersson, M. Andersson, M. Bergman, V. Blomqvist, and T. Martin. Student paper: HaskellDB improved. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115. ACM, 2004.

[3] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[4] W. R. C. Drijfhout. On the potential of functional relational databases. 2011.

[5] Haskell.org. Functional programming, Apr. 1. http://www.haskell.org/haskellwiki/Functional_programming.

[6] Ichikawa Y. *Database States in Lazy Functional Programming Languages: Imperative Update and Lazy Retrieval*. PhD thesis, Ochanomizu University, 1995.

[7] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. *Haskell '04 Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107, 2004.

[8] D. Leijen, C. Andersson, M. Andersson, M. Bergman, V. Blomqvist, B. Bringert, A. Hockersten, T. Martin, J. Shaw, and J. Bailey. The haskelldb package, May 2011. http://hackage.haskell.org/package/haskelldb. Retrieved at 4 June 2011.

[9] G. Meehan and M. Joy. Compiling lazy functional programs to Java bytecode. *Software: Practice and Experience*, 29(7):617–645, 1999.

[10] S. L. Osborn and T. E. Heaven. The design of a relational database system with abstract data types for domains. *ACM Trans. Database Syst.*, 11:357–373, August 1986.

[11] S. Stanczyk. Functional Programming for Databases, June 2009. http://www.ssw.uni-linz.ac.at/Teaching/Lectures/SpezialLVA/Stanczyk/. Retrieved at 4 June 2011.