Compiling ER Specifications into Declarative Programs[⋆]

Bernd Braßel Michael Hanus Marion Müller

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
{bbr|mh|mam}@informatik.uni-kiel.de

Abstract. This paper proposes an environment to support high-level database programming in a declarative programming language. In order to ensure safe database updates, all access and update operations related to the database are generated from high-level descriptions in the entity-relationship (ER) model. We propose a representation of ER diagrams in the declarative language Curry so that they can be constructed by various tools and then translated into this representation. Furthermore, we have implemented a compiler from this representation into a Curry program that provides access and update operations based on a high-level API for database programming.

1 Motivation

Many applications in the real world need databases to store the data they process. Thus, programming languages for such applications must also support some mechanism to organize the access to databases. This can be done in a way that is largely independent on the underlying programming language, e.g., by passing SQL statements as strings to some database connection. However, it is well known that such a loose coupling is the source of security leaks, in particular, in web applications [16]. Thus, a tight connection or amalgamation of the database access into the programming language should be preferred.

In principle, logic programming provides a natural framework for connecting databases (e.g., see [5, 7]) since relations stored in a relational database can be considered as facts defining a predicate of a logic program. Unfortunately, the well-developed theory in this area is not accompanied by practical implementations. For instance, distributions of Prolog implementations rarely come with a standard interface to relational databases. An exception is Ciao Prolog that has a persistence module [4] that allows the declaration of predicates where the facts are persistently stored, e.g., in a relational database. This module supports a simple method to query the relational database, but updates are handled by predicates with side effects and transactions are not explicitly supported. A similar concept but with a clear separation between queries, updates, and transactions has been proposed in [11] for the multi-paradigm declarative language

⋆ This work was partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.

Curry [8, 15]. This will be the basis for the current framework that provides an environment for high-level programming with databases. The objectives of this work are:

- The methods to access and update the database should be expressed by language features rather than passing SQL strings around.
- Queries to the database should be clearly separated from updates that might change the outcome of queries.
- Safe transactions, i.e., sequence of updates that keep some integrity constraints, should be supported.
- The necessary code for these operations should be derived from specifications whenever possible in order to obtain more reliable applications.

For this purpose, we define an API for database programming in Curry that abstracts from the concrete methods to access a given database by providing abstract operations for this purpose. In particular, this API exploits the type system of Curry in order to ensure a strict separation between queries and updates. This is described in detail in Section 2. To specify the logical structure of the data to be stored in a database, we use the entity-relationship (ER) model [3], which is well established for this purpose. In order to be largely independent of concrete specification tools, we define a representation of ER diagrams in Curry so that concrete ER specification tools can be connected by defining a translator from the format used in these tools into this Curry representation. This representation is described in Section 3. Finally, we develop a compiler that translates an ER specification into a Curry module that contains access and update operations and operations to check integrity constraints according to the ER specification. The generated code uses the database API described in Section 2. The compilation method is sketched in Section 4. Finally, Section 5 contains our conclusions.

2 Database Programming in Curry

We assume basic familiarity with functional logic programming (see [13] for a recent survey) and Curry [8, 15] so that we give in the following only a short sketch of the basic concepts relevant for this paper.

Functional logic languages integrate the most important features of functional and logic languages to provide a variety of programming concepts to the programmer. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming are combined with logic programming features like computing with partial information (logic variables), constraint solving, and non-deterministic search for solutions. This combination, supported by optimal evaluation strategies [1] and new design patterns [2], leads to better abstractions in application programs such as implementing graphical user interfaces [9], programming dynamic web pages [10, 12], or access and manipulation of persistent data possibly stored in databases [6, 11].

As a concrete functional logic language, we use Curry in our framework but it should be possible to apply the same ideas also to other functional logic languages, e.g., TOY [17]. From a syntactic point of view, a Curry program is a functional program extended by the possible inclusion of free (logic) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [18], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$"). A Curry *program* consists of the definition of functions and data types on which the functions operate. Functions are first-class citizens as in Haskell and are evaluated lazily. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. Function calls with free variables are evaluated by a possibly nondeterministic instantiation of demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule.

Example 1. The following Curry program defines the data types of Boolean values, "possible" (maybe) values, union of two types, and polymorphic lists (first four lines) and functions for computing the concatenation of lists and the last element of a list:

```
data Bool       = True    | False
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
data List a     = []      | a : List a

conc :: [a] -> [a] -> [a]
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last :: [a] -> a
last xs | conc ys [x] =:= xs   = x   where x,ys free
```

The data type declarations define `True` and `False` as the Boolean constants, `Nothing` and `Just` as the constructors for possible values (where `Nothing` is considered as no value), `Left` and `Right` to inject values into a union (`Either`) type, and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` and `b` are type variables ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell).

Curry also offers other standard features of functional languages, like higher-order functions (e.g., "`\`x` -> `e" denotes an anonymous function that assigns to each x the value of e), modules, or monadic I/O [19]. For instance, an operation of type "`IO t`" is an I/O action, i.e., a computation that interacts with the "external world" and returns a value of type `t`. Thus, purely declarative computations are distinguished from I/O actions by their types so that they cannot be freely mixed.

Logic programming is supported by admitting function calls with free variables (see "`conc ys [x]`" above) and constraints in the condition of a defining rule. Conditional program rules have the form $l \mid c \; = \; r$ specifying that l is reducible to r if the condition c is satisfied (see the rule defining `last` above). A *constraint* is any expression of the built-in type `Success`. For instance, the trivial constraint `success` is an expression of type `Success` that denotes the always satisfiable constraint. "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2, i.e., this expression is evaluated by proving both argument constraints concurrently. An *equational constraint* e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable constructor terms. Specific Curry systems also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints (e.g., PAKCS [14]).

Using functions instead of predicates has the advantage that the information provided by functional dependencies can be used to reduce the search space and evaluate goals in an optimal way (e.g., shortest derivation sequences, minimal solution sets, see [1] for details). However, there are also situations where a relational style is preferable, e.g., for database applications as considered in this paper. This style is supported by considering predicates as functions with result type `Success`. For instance, a predicate `isPrime` that is satisfied if the argument (an integer number) is a prime can be modeled as a function with type

```
isPrime :: Int -> Success
```

The following rules define a few facts for this predicate:

```
isPrime 2 = success
isPrime 3 = success
isPrime 5 = success
isPrime 7 = success
```

Apart from syntactic differences, any pure logic program has a direct correspondence to a Curry program. For instance, a predicate `isPrimePair` that is satisfied if the arguments are primes that differ by 2 can be defined as follows:

```
isPrimePair :: Int -> Int -> Success
isPrimePair x y = isPrime x & isPrime y & x+2 =:= y
```

In order to deal with information that is persistently stored outside the program (e.g., in databases), [11] proposed the concept of dynamic predicates. A *dynamic predicate* is a predicate where the defining facts (see `isPrime`) are not part of the program but stored outside. Moreover, the defining facts can be modified (similarly to dynamic predicates in Prolog). In order to distinguish between definitions in a program (that do not change over time) and dynamic entities, there is a distinguished type `Dynamic` for the latter.[1] For instance, in order to define a dynamic predicate `prime` to store prime numbers whenever we compute them, we provide the following definition in our program:

```
prime :: Int -> Dynamic
prime dynamic
```

[1] In contrast to Prolog, where dynamic declarations are often used for efficiency purposes, this separation is also necessary here due to the lazy evaluation strategy which makes it difficult to estimate *when* a particular evaluation is performed. Thus, performing updates by implicit side effects is not a good choice.

If the prime numbers should be persistently stored, one has to replace the second line by

```
prime persistent "store"
```

where *store* specifies the storage mechanism, e.g., a directory for a lightweight file-based implementation or a database specification [6].

There are various primitives that deal with dynamic predicates. First, there are combinators to construct complex queries from basic dynamic predicates. For instance, the combinator

```
(<>) :: Dynamic -> Dynamic -> Dynamic
```

joins two dynamic predicates, and the combinators

```
(|>)  :: Dynamic -> Bool    -> Dynamic
(|&>) :: Dynamic -> Success -> Dynamic
```

restrict a dynamic predicate with a Boolean condition or constraint, respectively. Since the operator "`<>`" binds stronger then "`|>`", the expression

```
prime x <> prime y |> x+2==y
```

specifies numbers `x` and `y` that are prime pairs.[2] On the one hand, such expressions can be translated into corresponding SQL statements [6] so that the programmer is freed of dealing with details of SQL. On the other hand, one can use all elements and libraries of a universal programming language for database programming due to its conceptual embedding in the programming language.

Since the contents of dynamic predicates can change over time, one needs a careful concept of evaluating dynamic predicates in order to keep the declarative style of programming. For this purpose, we introduce a concept of queries that are evaluated in the I/O monad, i.e., at particular points of time in a computation.[3] Conceptually, a *query* is a method to compute solutions w.r.t. dynamic predicates. Depending on the number of requested solutions, there are different operations to construct queries, e.g.,

```
queryAll :: (a -> Dynamic) -> Query [a]
queryOne :: (a -> Dynamic) -> Query (Maybe a)
```

`queryAll` and `queryOne` construct queries to compute all and one (if possible) solution to an abstraction over dynamic predicates, respectively. For instance,

```
qPrimePairs :: Query [(Int,Int)]
qPrimePairs = queryAll (\(x,y) -> prime x <> prime y |> x+2==y)
```

is a query to compute all prime pairs. In order to access the currently stored data, there is an operation `runQ` to execute a query as an I/O action:

```
runQ :: Query a -> IO a
```

[2] Since the right argument of "`|>`" demands a Boolean value rather than a constraint, we use the Boolean equality operator "`==`" rather than the equational constraint "`=:=`" to compare the primes `x` and `y`.

[3] Note that we only use the basic concept of dynamic predicates from [11]. The following interface to deal with queries and transactions is new and more abstract than the concepts described in [11].

For instance, executing the main expression "`runQ qPrimePairs`" returns prime pairs that can be derived from the prime numbers currently stored in the dynamic predicate `prime`.

In order to change the data stored in dynamic predicates, there are operations to add and delete knowledge about dynamic predicates:

```
addDB    :: Dynamic -> Transaction ()
deleteDB :: Dynamic -> Transaction ()
```

Typically, these operations are applied to single ground facts (since facts with free variables cannot be persistently stored), like "`addDB (prime 13)`" or "`deleteDB (prime 4)`". In order to embed these update operations into safe transactions, the result type is "`Transaction ()`" (in contrast to the proposal in [11] where these updates are I/O actions). A *transaction* is basically a sequence of updates that is completely executed or ignored (following the ACID principle in databases). Similarly to the monadic approach to I/O, transactions also have a monadic structure so that transactions can be sequentially composed by a monadic bind operator:

```
(|>>=) :: Transaction a -> (a -> Transaction b) -> Transaction b
```

Thus, "`t1 |>>= \x -> t2`" is a transaction that first executes transaction `t1`, which returns some result value that is bound to the parameter `x` before executing transaction `t2`. If the result of the first transaction is not relevant, one can also use the specialized sequential composition "`|>>`":

```
(|>>) :: Transaction a -> Transaction b -> Transaction b
t1 |>> t2 = t1 |>>= \_ -> t2
```

A value can be mapped into a trivial transaction returning this value by the usual return operator:

```
returnT :: a -> Transaction a
```

In order to define a transaction that depend on some data stored in a database, one can also embed a query into a transaction:

```
getDB :: Query a -> Transaction a
```

For instance, the following expression exploits the standard higher-order functions `map`, `foldr`, and "`.`" (function composition) to define a transaction that deletes all known primes that are smaller than `100`:

```
getDB (queryAll (\i -> prime i |> i<100)) |>>=
foldr (|>>) (returnT ()) . map (deleteDB . prime)
```

Since such a sequential combination of transactions that are the result of mapping a list of values into a list of transactions frequently occurs, there is also a single function for this combination:

```
mapT_ :: (a -> Transaction _) -> [a] -> Transaction ()
mapT_ f = foldr (|>>) (returnT ()) . map f
```

To apply a transaction to the current database, there is an operation `runT` that executes a given transaction as an I/O action:

```
runT :: Transaction a -> IO (Either a TError)
```

`runT` returns either the value computed by the successful execution of the transaction or an error in case of a transaction failure. The type `TError` of possible transaction errors contains constructors for various kinds of errors, i.e., it is currently defined as

```
data TError = TError TErrorKind String

data TErrorKind = KeyNotExistsError | DuplicateKeyError
  | UniqueError | MinError | MaxError | UserDefinedError
```

but this type might be extended according to future requirements (the string argument is intended to provide some details about the reason of the error). `UserDefinedError` is a general error that could be raised by the application program whereas the other alternatives are typical errors due to unsatisfied integrity constraints. An error is raised inside a transaction by the operation

```
errorT :: TError -> Transaction a
```

where the specialization

```
failT :: String -> Transaction a
failT s = errorT (TError UserDefinedError s)
```

is useful to raise user-defined transaction errors. If an error is raised inside a transaction, the transaction is aborted, i.e., the transaction monad satisfies the laws

$$\begin{aligned} \texttt{errorT}\ e\ \texttt{|>>=}\ t &= \texttt{errorT}\ e \\ t\ \texttt{|>>=}\ \texttt{\textbackslash}x\ \texttt{->}\ \texttt{errorT}\ e &= \texttt{errorT}\ e \\ \texttt{runT (errorT}\ e\texttt{)} &= \texttt{return (Right}\ e\texttt{)} \end{aligned}$$

Thus, the changes to the database performed in a transaction that raises an error are not visible.

There are a few further useful operations on transactions that are not relevant for this paper so that we omit them. We summarize the important features of this abstract programming model for databases:

- Persistent data is represented in the application program as language entities (i.e., dynamic predicates) so that one can use all features of the underlying programming language (e.g., recursion, higher-order functions, deduction) for programming with this data.
- There is a clear separation between the data access (i.e., queries) and updates that can influence the results of accessing data. Thus, queries are purely declarative and are applied to the actual state of the database when their results are required.
- Transactions, i.e., database updates, can be constructed from a few primitive elements by specific combinators. Transactions are conceptually executed as an atomic action on the database. Transactions can be sequentially composed but nested transactions are excluded due to the type system (this feature is intended since nested transactions are usually not supported in databases).

All features for database programming are summarized in a specific `Database` library[4] so that they can be simply used in the application program by importing

[4] `http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/Database.html`

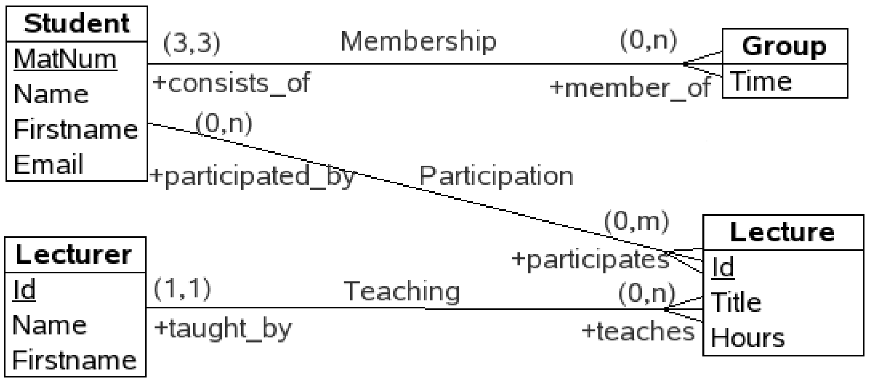

Fig. 1. A simple entity-relationship diagram for university lectures

it. This will be the basis to generate higher-level code from entity-relationship diagrams that are described in the following.

3 Entity-Relationship Diagrams

The entity-relationship model is a framework to specify the structure and specific constraints of data stored in a database. It uses a graphical notation, called entity-relationship diagrams (ERDs) to visualize the conceptual model. In this framework, the part of the world that is interesting for the application is modeled by entities that have attributes and relationships between the entities. The relationships have cardinality constraints that must be satisfied in each valid state of the database, e.g., after each transaction.

There are various tools to support the data modeling process with ERDs. In our framework we want to use some tool to develop specific ERDs from which the necessary program code based on the `Database` library described in the previous section can be automatically generated. In order to become largely independent of a concrete tool, we define a representation of ERDs in Curry so that a concrete ERD tool can be applied in this framework by implementing a translator from the tool format into our representation. In our concrete implementation, we have used the free software tool Umbrello UML Modeller[5], a UML tool part of KDE that also supports ERDs. Figure 1 shows an example ERD constructed with this tool. The developed ERDs are stored in XML files in XMI (XML Metadata Interchange) format, a format for the exchange of UML models. Thus, it is a standard XML transformation task to translate the Umbrello format into our ERD format.

The representation of ERDs as data types in Curry is straightforward. A complete ERD consists of a name (that is later used as the module name for the generated code) and lists of entities and relationships:

[5] `http://uml.sourceforge.net`

```
data ERD = ERD String [Entity] [Relationship]
```

An entity has a name and a list of attributes, where each attribute has a name, a domain, and specifications about its key and null value property:

```
data Entity = Entity String [Attribute]

data Attribute = Attribute String Domain Key Null

data Key = NoKey | PKey | Unique

type Null = Bool

data Domain = IntDom             (Maybe Int)
            | FloatDom           (Maybe Float)
            | CharDom            (Maybe Char)
            | StringDom          (Maybe String)
            | BoolDom            (Maybe Bool)
            | DateDom            (Maybe ClockTime)
            | UserDefined String (Maybe String)
            | KeyDom String   -- later used for foreign keys
```

Thus, each attribute is part of a primary key (`PKey`), unique (`Unique`), or not a key (`NoKey`). Furthermore, it is allowed that specific attributes can have null values, i.e., can be undefined. The domain of each attribute is one of the standard domains or some user-defined type. In the latter case, the first argument of the constructor `UserDefined` is the qualified type name used in the Curry application program (note that the `Database` library is able to handle complex types by mapping them into standard SQL types [6]). For each kind of domain, one can also have a default value (modeled by the `Maybe` type in Curry). The constructor `KeyDom` is not necessary to represent ERDs but will be later used to transform ERDs into relational database schema.

Finally, each relationship has a name and a list of connections to entities (`REnd`), where each connection has the name of the connected entity, the role name of this connection, and its cardinality as arguments:

```
data Relationship = Relationship String [REnd]

data REnd = REnd String String Cardinality

data Cardinality = Exactly Int | Range Int (Maybe Int)
```

The cardinality is either a fixed integer or a range between two integers (where `Nothing` as the upper bound represents an arbitrary cardinality). For instance, the simple-complex (1:n) relationship `Teaching` in Figure 1 can be represented by the term

```
Relationship "Teaching"
             [REnd "Lecturer" "taught_by" (Exactly 1),
              REnd "Lecture" "teaches" (Range 0 Nothing)]
```

4 Compiling ER Diagrams into Curry Programs

This section describes the transformation of ERDs into executable Curry code. This transformation is done in the following order:

1. Translate an ERD into an `ERD` term.
2. Represent the relationships occurring in an `ERD` term as entities.
3. Map all entities into corresponding Curry code based on the `Database` library.

The first step depends on the format used in the ERD tool. As mentioned above, we have implemented a translation from the XMI format used by the Umbrello UML Modeller into `ERD` terms. This part is relatively easy thanks to the presence of XML processing tools.

4.1 Transforming ERDs

The second step is necessary since the relational model supports only relations (i.e., database tables). Thus, entities as well as relationships must be mapped into relations. The mapping of entities into relations is straightforward by using the entity name as the name of the relation and the attribute names as column names. The mapping of relationships is more subtle. In principle, each relationship can be mapped into a corresponding relation. However, this simple approach might cause the creation of many relations or database tables. In order to reduce them, it is sometimes better to represent specific relations as foreign keys, i.e., to store the key of entity e_1 referred by a relationship between e_1 and e_2 in entity e_2. Whether or not this is possible depends on the kind of the relation. The different cases will be discussed next. Note that the representation of relationships as relations causes also various integrity constraints to be satisfied. For instance, if an entity has an attribute which contains a foreign key, the value of this attribute must be either null or an existing key in the corresponding relation. Furthermore, the various cardinalities of each relationship must be satisfied. Ideally, each transaction should ensure that the integrity constraints are valid after finishing the transaction.

Now we discuss the representation of the various kinds of relationships in the ER model. For the sake of simplicity, we assume that each relationship contains two ends, i.e., two roles with cardinality ranges (min, max) so that we can characterize each relationship by their related cardinalities $(min_A, max_A) : (min_B, max_B)$ between entities A and B (where max_i is either a natural number greater than min_i or ∞, $i \in \{A, B\}$).

Simple-simple (1:1) relations: This case covers all situations where each cardinality is at most one. In the case $(0, 1) : (1, 1)$, the key of entity B is added as an attribute to entity A containing a foreign key since there must be exactly one B entity for each A entity. Furthermore, this attribute is `Unique` to ensure the uniqueness of the inverse relation. The case $(0, 1) : (0, 1)$ can be similarly treated except that null values are allowed for the foreign key.

Simple-complex (1:n) relations: In the case $(0, 1) : (min_B, max_B)$, the key of entity A is added as a foreign key (possibly null) to each B entity. If $min_B > 0$ or $max_B \neq \infty$, the integrity constraints for the right number of occurrences must be checked by each database update. The case $(1, 1)$:

$(0, max_B)$ is similarly implemented except that null values for the foreign key are not allowed.

Complex-complex (n:m) relations: In this case a new relation representing this relationship is introduced. The new relation is connected to entities A and B by two new relationships of the previous kinds.

Note that we have not considered relationships where both minimal cardinalities are greater than zero. This case is excluded by our framework (and rarely occurs in practical data models) since it causes difficulties when creating new entities of type A or B. Since each entity requires a relation to an existing entity of the other type and vice versa, it is not possible to create the new entities independently. Thus, both entities must be created and connected in one transaction which requires specific complex transactions. Therefore, we do not support this in our code generation. If such relations are required in an application (e.g., cyclic relationships), then the necessary code must be directly written with the primitives of the `Database` library.

Based on this case distinction, the second step of our compiler maps an `ERD` term into a new `ERD` term where foreign keys are added to entities and new entities are introduced to represent complex-complex relations. Furthermore, each entity is extended with an internal primary key to simplify the access to each entity by a unique scheme.

4.2 Code Generation for ERDs

After the mapping of entities and relationships into relations as described above, we can generate the concrete program code to organize the database access and update. As already mentioned, we base the generated code on the functionality provided by the library `Database` described in Section 2. The schemas for the generated code are sketched in this section. We use the notation *En* for the name of an entity (which starts by convention with an uppercase letter) and *en* for the same name where the first letter is lowercase (this is necessary due to the convention in Curry that data constructors and functions start with uppercase and lowercase letters, respectively).

The first elements of the generated code are data types to represent relations. For each entity *En* with attributes of types $at_1, \ldots, at_n$, we generate the following two type definitions:

`data` *En* `=` *En* `Key` $at_1 \ldots at_n$

`data` *En*`Key =` *En*`Key Key`

`Key` is the type of all internal keys for entities. Currently, it is identical to `Int`. Thus, each entity structure contains an internal key for its unique identification. The specific type *En*`Key` is later used to distinguish the keys for different entities by their types, i.e., to exploit the type system of Curry to avoid confusion between the various keys. For each relation that has been introduced for a complex-complex relationship (see above), a similar type definition is introduced except that it does not have an internal key but only the keys of the connected entities as arguments. Note that only the names of the types are exported but not their internal structure (i.e., they are *abstract* data types for the application program). This ensures that the application program cannot manipulate the internal keys. The manipulation of attributes is possible by explicit getter and setter functions that are described next.

In order to access or modify the attributes of an entity, we generate corresponding functions where we use the attribute names of the ERD for the names of the functions. If entity *En* has an attribute A_i of type at_i ($i = 1, \ldots, n$), we generate the following getter and setter functions and a function to access the key of the entity:

```
enAi :: En -> ati
enAi (En _ ... xi ... _) = xi

setEnAi :: En -> ati -> En
setEnAi (En x1 ... _ ... xn) xi = En x1 ... xi ... xn

enKey :: En -> EnKey
enKey (En k _ ... _) = EnKey k
```

As described in Section 2, data can be persistently stored by putting them into a dynamic predicate. Thus, we define for each entity *En* a dynamic predicate

```
enEntry :: En -> Dynamic
enEntry persistent "..."
```

Since the manipulation of all persistent data should be done by safe operations, this dynamic predicate is not exported. Instead, a dynamic predicate *en* is exported that associates a key with the data so that an access is only possible to data with an existing key:

```
en :: EnKey -> En -> Dynamic
en key obj | key=:=enKey obj = enEntry obj
```

Although these operations seem to be standard functions, the use of a functional logic language is important here. For instance, the access to an entity with a given key `k` can be done by solving the goal "*en* `k o`" where `o` is a free variable that will be bound to the concrete instance of the entity.

For each role with name *rn* specified in an ERD, we also generate a dynamic predicate of type

```
rn :: En1Key -> En2Key -> Dynamic
```

where En_1 and En_2 are the entities related by this role. The implementation of these predicates depend on the kind of relationship according to their implementation as discussed in Section 4.1. Since complex-complex relationships are implemented as relations, i.e., persistent predicates (that are only internal and not exported), the corresponding roles can be directly mapped to these. Simple-simple and simple-complex relationships are implemented by foreign keys in the corresponding entities. Thus, their roles are implemented by accessing these keys. We omit the code details that depend on the different cases already discussed in Section 4.1.

Based on these basic implementations of entities and relationships, we also generate code for transactions to manipulate the data and check the integrity constraints specified by the relationships of an ERD. In order to access an entity with a specific key, there is a generic function that delivers this entity in a transaction or raises a transaction error if there is no entry with this key:

```
getEntry :: k -> (k -> a -> Dynamic) -> Transaction a
getEntry key pred =
  getDB (queryOne (\info -> pred key info)) |>>=
  maybe (errorT (KeyNotExistsError "no entry for..."))
        returnT
```

This internal function is specialized to an exported function for each entity:

```
getEn :: EnKey -> Transaction En
getEn key = getEntry key en
```

In order to insert new entities, there is a "new" transaction for each entity. If the ERD specifies no relationship for this entity with a minimum greater than zero, there is no need to provide related entities so that the transaction has the following structure (if *En* has attributes of types $at_1, \ldots, at_n$):

```
newEn :: at1 -> ··· -> atn -> Transaction En
newEn a1 ... an = check1 |>> ... |>> checkn |>> newEntry ...
```

Here, $check_i$ are the various integrity checks (e.g., uniqueness checks for attributes specified as `Unique`) and `newEntry` is a generic operation (similarly to `getEntry`) to insert a new entity. If attribute A_i has a default value or null values are allowed for it, the type at_i is replaced by `Maybe` at_i in `new`*En*. If there are relationships for this entity with a minimum greater than zero, than the keys (in general, a list of keys) must be also provided as parameters to `new`*En*. The same holds for the "new" operations generated for each complex-complex relationship. For instance, the new operation for lectures according to the ERD in Figure 1 has the following type:

```
newLecture :: LecturerKey -> Int -> String -> Maybe Int
              -> Transaction Lecture
```

The first argument is the key of the lecturer required by the relationship `Teaching`, and the further arguments are the values of the `Id`, `Title` and `Hours` attributes (where the attribute `Hours` has a default value so that the argument is optional).

Similarly to `new`*En*, we provide also operations to update existing entities. These operations have the following structure:

```
updateEn :: En -> Transaction ()
updateEn e = check1 |>> ... |>> checkn |>> updateEntry ...
```

Again, the various integrity constraints must be checked before an update is finally performed. In order to get an impression of the kind of integrity constraints, we discuss a few checks in the following.

For instance, if an attribute of an entity is `Unique`, this property must be checked before a new instance of the entity is inserted. For this purpose, there is a generic transaction

```
unique :: a -> (b -> a) -> (b -> Dynamic) -> Transaction ()
```

where the first argument is the attribute value, the second argument is a getter function for this attribute, and the third argument is the dynamic predicate representing this entity, i.e., a typical call to check the uniqueness of the new value a_i for attribute A_i of entity *En* is (`unique` a_i enA_i En). This transaction raises a `UniqueError` if an instance with this attribute value already exists.

If an entity contains a foreign key, each update must check the existence of this foreign key. This is the purpose of the generic transaction

```
existsDBKey :: k -> (a -> k) -> (a -> Dynamic) -> Transaction ()
```

where the arguments are the foreign key, a getter function (*en*`Key`) for the key in the foreign entity and the dynamic predicate of the foreign entity. If the key does not exist, a `KeyNotExistsError` is raised. Furthermore, there are generic transactions to check minimum and maximum cardinalities for relationships and lists of foreign keys that can raise the transaction errors `MinError`, `MaxError`, or `DuplicateKeyError`. For each `new` and `update` operation generated by our compiler, the necessary integrity checks are inserted based on the ER specification.

Our framework does not provide delete operations. The motivation for this is that safe delete operations require the update of all other entities where this entity could occur as a key. Thus, a simple delete could cause many implicit changes that are difficult to overlook. It might be better to provide only the deletion of single entities followed by a global consistency check (discussed below). A solution to this problem is left as future work.

Even if our generated transactions ensure the integrity of the affected relations, it is sometimes useful to provide a global consistency check that is regularly applied to all data. This could be necessary if unsafe delete operations are performed, or the database is modified by programs that do not use the safe interface but directly accesses the data. For this purpose, we also generate a global consistency test that checks all persistent data w.r.t. the ER model. If $E_1, \ldots, E_n$ are all entities (including the implicit entities for complex-complex relations) derived from the given ERD, the global consistency test is defined by

```
checkAllData :: Transaction ()
checkAllData = checkE1 |>> ... |>> checkEn
```

The consistency test for each entity *En* is defined by

```
checkEn :: Transaction ()
checkEn = getDB (queryAll enEntry) |>>= mapT_ checkEnEntry

checkEnEntry :: En -> Transaction ()
checkEnEntry e = check1 |>> ... |>> checkn
```

where the tests $check_i$ are similar to the ones used in new and update operations that raise transaction errors in case of unsatisfied integrity constraints.

5 Conclusions

We have presented a framework to compile conceptual data models specified as entity-relationship diagrams into executable code for database programming in Curry. This compilation is done in three phases: translate the specific ERD format into a tool-independent representation, transform the relationships into relations according to their complexity, and generate code for the safe access and update of the data.

Due to the importance of ERDs to design conceptual data models, there are also other tools with similar objectives. Most existing tools support only the generation of SQL code, like the free software tools DB-Main[6] or DBDesigner4[7]. The main motivation for our development was the seamless embedding of database programming in a declarative programming language and the use of existing specification methods like ERDs as the basis to generate most of the necessary code required by the application programs. The advantages of our framework are:

- The application programmer must only specify the data model in a high-level format (ERDs) and all necessary code for dealing with data in this model is generated.
- The interface used by the application programs is type safe, i.e., the types specified in the ERD are mapped into types of the programming language so that ill-typed data cannot be constructed.
- Updates to the database are supported as transactions that automatically checks all integrity constraints specified in the ERD.
- Checks for all integrity constraints are derived from the ERD for individual tables and the complete database so that they can be periodically applied to verify the integrity of the current state of the database.
- The generated code is based on an abstract interface for database programming so that it is readable and well structured. Thus, it can be easily modified and adapted to new requirements. For instance, integrity constraints not expressible in ERDs can be easily added to individual update operations, or specific deletion operations can be inserted in the generated module.

For future work we intend to increase the functionality of our framework, e.g., to extend ERDs by allowing the specification of more complex integrity constraints or attributes for relations, which is supported by some ER tools, or to provide also delete operations for particular entities. Finally, it could be also interesting to generate access and update operations for existing databases by analyzing their data model. Although this is an issue different from our framework, one can reuse the API described in Section 2 and some other techniques of this paper for such a purpose.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.

[6] `http://www.db-main.be`
[7] `http://www.fabforce.net/dbdesigner4`

2. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.
3. P. P.-S. Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1, No. 1, pp. 9–36, 1976.
4. J. Correas, J.M. Gómez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations). In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pp. 104–119. Springer LNCS 3057, 2004.
5. S.K. Das. *Deductive Databases and Logic Programming.* Addison-Wesley, 1992.
6. S. Fischer. A Functional Logic Database Library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 54–59. ACM Press, 2005.
7. H. Gallaire and J. Minker, editors. *Logic and Databases*, New York, 1978. Plenum Press.
8. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
9. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
10. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
11. M. Hanus. Dynamic Predicates in Functional Logic Programs. *Journal of Functional and Logic Programming*, Vol. 2004, No. 5, 2004.
12. M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pp. 27–38. ACM Press, 2006.
13. M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.
14. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2006.
15. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.
16. S.H. Huseby. *Innocent Code: A Security Wake-Up Call for Web Programmers.* Wiley, 2003.
17. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
18. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.
19. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.