Lazy Database Access with Persistent Predicates[⋆]

Sebastian Fischer

Institut für Informatik
Universität Kiel
Olshausenstraße 40, 24098 Kiel, Germany

Abstract

Programmers need mechanisms to store application specific data that persists multiple program runs. To accomplish this task, they usually have to deal with storage specific code to access files or relational databases. Functional logic programming provides a natural framework to transparent persistent storage through persistent predicates, i.e., predicates with externally stored facts.
We extend previous work on persistent predicates for Curry by lazy database access. Results of a database query are only read as much as they are demanded by the application program. We also present a type-oriented approach to convert between database and Curry values which is used to implement lazy access to persistent predicates based on a low level lazy database interface.

Keywords: Curry, database access, dynamic predicates, laziness, persistence

1 Introduction

Programming languages need mechanisms to store data that persists among program executions. Internal data needs to be saved and recovered, and external data has to be represented and manipulated by an application. For instance, web applications often read data stored on a database server and present it to the user in a structured way.

Relational databases are typically used to efficiently access a large amount of stored data. In a relational database, data is stored in tables that can be divided into rows and columns. From the logic programming point of view, a database table can be seen as specification of a predicate, storing the predicate's facts in its rows. In previous work [7,5,4], we developed an approach to database access in Curry where database tables are seen as dynamic specification of special predicates. The specification is dynamic because it may change at run time. Hence, such predicates are called *dynamic predicates*. Dynamic predicates whose facts are stored persistently, e.g., in a database, are called *persistent predicates*.

⋆ This work has been partially supported by the DFG under grant Ha 2457/5-1.

A first attempt of a prototypical implementation of our approach turned out to be inefficient for large result sets because these were parsed completely when a query was executed. Therefore, we developed a lazy implementation that does not read unused results. This implementation consists of two parts:

- we develop a low level lazy database interface and
- we implement lazy access to persistent predicates based on this interface.

The most interesting point of the second part is concerned with data conversion. We present a type-oriented approach to convert between the values stored in a database and Curry data terms. This paper mainly describes practical work. Although we slightly modify the interface of our library, we do not introduce conceptual novelties. Nevertheless, it is remarkable that we could employ high-level declarative techniques to achieve quite technical goals. As a result, we get a concise and both portable and maintainable implementation.

The remainder of this paper is structured as follows: in Section 1.1 we motivate persistent predicates by reflecting related work on database access in declarative programming languages. In Section 2 we present the interface of our database library. We discuss a low level implementation of lazy database access in Section 3. We sketch a type-oriented approach to conversion between database and Curry values in Section 4 that is used to implement lazy access to persistent predicates based on the low level interface. Finally, Section 5 contains concluding remarks.

1.1 Related Work

The notion of persistent predicates is introduced in [3] where a database and a file based Prolog implementation are provided. Persistent predicates enable the programmer to store data that persists from one execution to the next and is stored transparently, i.e., the program's source code need not be changed with the storage mechanism. However the implementation presented in [3] has two major drawbacks: firstly, access to persistent predicates is implemented using side effects. This is a problem, because in this approach the behavior of a program with calls to persistent predicates depends on the order of evaluation. Another drawback of [3] is that it, secondly, does not support transactions. Databases are often used in the context of web applications where potentially a lot of processes access the database concurrently. Therefore, transactions are a key feature for a practical implementation of a database library.

Both problems are solved in [7] where database access is only possible inside the IO monad [11] and a transaction concept is provided. Eliminating side effects is especially essential in the context of functional logic programming languages which are based on sophisticated evaluation strategies [1]. [4] extends the library presented in [7] by a database implementation of persistent predicates. We improve this implementation by means of lazy database access and slightly simplify its interface. Section 2 recapitulates the interface to our library and mentions differences to the version presented in [4].

A combinator library for Haskell, which is used to construct database queries with relational algebra, is provided by [9]. It allows for a syntactically correct and type safe implementation of database access. The authors provide a general approach to embed domain specific languages into higher-order typed languages and apply it to relational algebra to access relational databases. While [9] *syntactically* integrates a domain specific language into the Haskell programming language, persistent predicates *transparently* integrate database access into a familiar programming paradigm.

2 The Database Library

In our approach, persistent predicates are defined by the keyword `persistent`, since their definition is not part of the program but externally stored. The only information given by the programmer is a type signature and a string argument to `persistent` identifying the storage location. The predicate defined below stores (finitely many) prime numbers in the table `primes` in the database `currydb`:

```
prime :: Int -> Dynamic
prime persistent "db:currydb.primes"
```

The storage location is prefixed with `"db:"` to indicate that it is a database table. After the colon, the database and the table are given divided by a period.

The result type of persistent predicates is `Dynamic` which is conceptually similar to `Success` (the result type of constraints and ordinary predicates in Curry.) Dynamic predicates are distinguished from other predicates to ensure that the functions provided to access them are only used for dynamic predicates, and not for ordinary ones. Conceptually, the type `Dynamic` should be understood as a datatype with a single value – just like `Success` is a datatype with the single value `success`. Internally, the datatype stores information on how to access the externally stored facts. This information is introduced by the predicate specifications and the combinators presented in Section 2.2.

2.1 Basic Operations

The basic operations for persistent predicates stored in a database are assertion to insert new facts, retraction to delete and query to retrieve them. Because the definition of persistent predicates changes over time, their access is only possible inside the IO monad to provide an explicit order of evaluation. To manipulate the facts of a persistent predicate, the operations

```
assert  :: Dynamic -> IO ()
retract :: Dynamic -> IO ()
```

are provided. Conceptually, these functions modify a global knowledge base as a side effect: `assert` inserts new facts into the knowledge base and `retract` removes them. The arguments of `assert` and `retract` must not contain free variables, and thus, only assertion and retraction of ground facts are allowed. If the arguments of a database predicate are not ground, a call to `assert` or `retract` suspends until they are. Note that, currently, Curry only supports concurrency of constraints via the concurrent conjunction operator

```
(&) :: Success -> Success -> Success
```

An extension of Curry similar to Concurrent Haskell [10] that supports concurrent i/o actions could make use of this feature for synchronization. We could define an alternative version of `retract` that does not suspend on partially instantiated arguments but deletes all matching facts from the database without propagating the resulting bindings to the program. These bindings would then be encapsulated in the call to `retract`. However, this behavior can be achieved using `getDynamicSolutions` defined below and, more importantly, the implementation of `retract` would become inefficient, if the encapsulation would be done internally for all calls to retract, i.e., also for those that do not involve free variables. Moreover, a similar alternative does not seem to exist for the implementation of `assert`. Representing unknown parts as *null*-values seems to be appropriate at first glance. However, the information about which variables are identical is lost, if all free variables are represented as *null*-values, thus, suspension or failure seem to be more practical for `retract` and the only reasonable options for `assert`. We chose suspension to support possible extensions of Curry concerned with concurrency.

A query to a persistent predicate can have multiple solutions computed nondeterministically. To encapsulate search, the function

```
getDynamicSolutions :: (a -> Dynamic) -> IO [a]
```

takes a dynamic predicate abstraction and returns a list of all values satisfying the abstraction similar to `getAllSolutions` for predicates with result type `Success`. The function

```
getDynamicSolution :: (a -> Dynamic) -> IO (Maybe a)
```

can be used to query only one solution. Note that this function would not be necessary if `getDynamicSolutions` were lazy. But with a strict implementation all solutions are computed in advance even if the program demands only the head of the result list. Unfortunately not all Curry implementations support lazy encapsulated search and we can provide it only for predicates that are stored in a database. Also note that only encapsulated access to a dynamic predicate is provided. Dynamic predicates cannot be used in guards like ordinary predicates and, thus, cannot cause nondeterministic behavior of the program.

2.2 Combining Persistent Predicates

Often information needs to be queried from more than one persistent predicate at once, or a query has to be restricted with a boolean condition. To combine several persistent predicates, we provide two different forms of conjunction. One combines two values of type `Dynamic` similarly to the function `(&)` for ordinary constraints. The other one combines a `Dynamic` predicate with a boolean condition:

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

These combinators can be employed to construct `Dynamic` abstractions that resemble typical database queries. For example, the abstraction

```
\(x,y) ->
 prime x <> prime y
 |> x+2 == y
```

resembles the SQL query

```
SELECT tab1.prime, tab2.prime
  FROM primes AS tab1, primes AS tab2
 WHERE tab1.prime + 2 = tab2.prime
```

The translation into SQL relies on very few primitive features present in SQL. Simple *select*-statements with a *where*-clause suffice to query facts of a persistent predicate – also if it involves complex conjunctions. Since there are no combinators to define predicates with aggregating arguments – like the sum, average, minimum or maximum of another argument – we do not need such features of SQL. Also, nested queries and *having*- or *order-by*-clauses are not generated by our implementation. We do not provide aggregation features because it is unclear how to perform `assert` on a predicate with aggregating arguments. Aggregation is only reasonable for queries and thus has to be coded explicitly. *Null*-values can be accessed as `Nothing` if the corresponding argument is of a `Maybe` type. If it is not, *null*-values are represented as free variables. A detailed transformation scheme including a mechanism to transform boolean conditions attached to persistent predicates into efficient SQL queries is discussed in [5,4].

2.3 Transactions

Since changes made to the definition of persistent predicates are instantly visible to other programs employing the same predicates, transactions are required to declare atomic operations. As database systems usually support transactions, the provided functions rely on the databases transaction support:

```
transaction      :: IO a -> IO (Maybe a)
abortTransaction :: IO a
```

The function `transaction` is used to start transactions. The given i/o action is performed atomically and `Nothing` is returned, if the i/o action fails or is explicitly aborted with `abortTransaction`. Nested transactions are not supported and lead to a run-time error. We simplified the interface of [4] by eliminating the function `transactionDB` that takes a database name to perform the transaction in. Now, transactions are specified with the function `transaction` regardless whether they employ database predicates or not and the transaction is performed in all databases known to the current process. Whether this is a performance penalty depends on the implementation of transactions in the involved database systems. Usually, a transaction that does not touch any tables, does not block other processes that access the database. Actually, one process will typically not access different database systems, so the simplified interface will rarely cause any performance overhead.

3 Lazy Database Access

Although complex restrictions can be expressed using the conjunction combinators `(<>)` and `(|>)` presented in the previous section, database queries may still have large result sets. If the programmer accesses only parts of the results, it is an unnecessary overhead to retrieve all results from the database.

The implementation presented in [4] communicates with the database via standard i/o and always parses the complete result set before providing it to the application program. This turned out to be inefficient for large result sets. Thus, we developed an alternative implementation. The implementation presented in this paper does not query the complete results when `getDynamicSolutions` is called. Instead, it only retrieves a *handle* which is used to query the results when they are demanded. The advantage of this approach is obvious: a call to `getDynamicSolutions` causes only a negligible delay because it only retrieves a handle instead of the whole result set from the database. Moreover, results that are not demanded by the application are not queried from the database. Thus, a delay is only caused for reading results that are consumed by the program – no results are queried in advance. If results are read on demand, it is important that they are independent of when they are demanded. Especially, results must not be affected by table updates that happen between the query and the consumption of the results. This property is ensured by the database system. Conceptually, a snapshot of the database is created when the query yields a handle and all results correspond to this snapshot. Hence, the fact that a query is read lazily does not affect the corresponding set of results.

Relational database systems allow us to retrieve result sets of queries incrementally via an API. In this section we show how we access this API from a Curry program. One possibility to access the database API is to use external functions. However, implementing them is a complex task and more importantly external functions need to be re-coded for every Curry implementation, so it is a good idea to avoid them wherever possible.

3.1 Curry Ports

Java supports a variety of database systems. Curry supports distributed programming using ports [6] and it is possible to port this concept to Java and write distributed applications that involve both Curry and Java programs. So we can access all database systems supported by Java in Curry if we can communicate with a Java program using Curry.

We implemented database access using ports to get a high-level, maintainable, and portable Curry implementation that benefits from the extensive support for different database systems in Java. We will not discuss how ports and database access are implemented in Java, but focus on the Curry part of our implementation. We will concentrate on ports and how we use them to model a lazy database interface.

A port is a multiset of messages which is constrained to hold exactly the elements of a specified list. There is a predicate

```
openPort :: Port a -> [a] -> Success
```

that creates a port for messages of type `a`. Usually, `openPort` is called with free variables as arguments and the second argument is instantiated by sending messages to the first argument. A client can send a message to a port using

```
send :: a -> Port a -> Success
```

Since the message may contain free variables that can be bound by the server, there is no need for a `receive` function on ports: if the client needs to receive an answer from the server, it can include a free variable in his request and wait for the server to bind this variable.

To share a port between different programs, it can be registered under a global name accessible over the network. The i/o action

```
openNamedPort :: String -> IO [a]
```

opens a globally accessible port and returns the list of messages that are sent to the port. The i/o action

```
connectPort :: String -> IO (Port a)
```

returns the port that is registered under the given name.

The last two functions destroy type-safety of port communication because their return values are polymorphic. The predicates `openPort` and `send` ensure that only type correct messages are sent to or received from a port. With `openNamedPort` and `connectPort`, however, it is possible to send type-incorrect messages to a port. The function `openNamedPort` creates a stream of messages of unspecified type and the type of messages that can be sent to a port created by `connectPort` is also unspecified. If ports are used for communication over a network, this communication is no longer type-safe. Therefore, we have to carefully establish type-correct message exchange ourselves. The user of our library is not concerned with these issues because the ports-based interface is not exported.

3.2 Lazy Interface to the Database

In this section we describe the messages that are used to communicate with the Java program that implements database access and the functions that use these messages to implement a lazy database interface. This interface is not intended for application programs but only used internally for our implementation of persistent predicates. For the communication with the Java program we need a datatype for the messages that are sent via ports and a datatype for the values that are stored in a database table. A Curry process must be able to open and close database connections and send *insert*-, *delete*-, or *commit*-statements. However the most interesting messages with regard to lazy database access are those concerned with queries and result retrieval. As mentioned earlier, a handle must be returned as result of a query. Furthermore, it must be possible to check, whether there are more results corresponding to a handle and if so to query another row of the result set. Hence, we define the following datatypes:

```
data DBMessage
  = Open String DBHandle
  | Update DBHandle String
  | Query DBHandle String ResultHandle
  | EndOfResults DBHandle ResultHandle Bool
  | NextRow DBHandle ResultHandle [DBValue]
  | Close DBHandle

type DBHandle = Int
type ResultHandle = Int

data DBValue
  = NULL | BOOLEAN Bool | INT Int
  | FLOAT Float | CLOB String | TIME ClockTime

type Connection = (Port DBMessage, DBHandle)
```

A connection consists of a port of type `(Port DBMessage)` and an integer of type `DBHandle`. The datatype `DBValue` wraps values of different SQL column types. SQL supports a variety of different column types and we chose a reasonable small representation as algebraic datatype. To obtain a small representation, we represent values of different SQL types as values of the same Curry type. For example, values of type `DOUBLE`, `FLOAT`, `REAL`, … are all represented as wrapped `Float` values in Curry. SQL supports the special datatypes `DATE`, `TIME` and `DATETIME` for date and time values that are represented as wrapped value of type `ClockTime` – the standard datatype for representing time values in Curry. Although subsuming column types is an abstraction, it is detailed enough for transparent database access in Curry.

A central part of our implementation is the definition of the datatype `DBMessage`. The defined messages serve the following purpose:

- `(Open` *spec db*`)` opens a new connection to the specified database and *db* is instantiated with an integer representing the connection.
- `(Update` *db sql*`)` performs an SQL statement *sql* in the database represented by *db* without returning a result.
- `(Query` *db sql result*`)` is used for queries that return a result. Note that only an integer that represents the result set is returned.
- `(EndOfResults` *db result empty*`)` checks whether the result set is empty,
- `(NextRow` *db result row*`)` queries one row from a non-empty result set and
- `(Close` *db*`)` closes the specified connection.

As a simple example for the communication with a database server over ports using this datatype consider the following definition:

```
endOfResults :: Connection -> ResultHandle -> Bool
endOfResults (p,db) r
  | send (EndOfResults db r empty) p = ensureNotFree empty
 where empty free
```

The call to `ensureNotFree` suspends until its argument is bound and is used to wait for the answer returned from the server. The function `endOfResults` and the `NextRow`-message can be employed to define a function

```
query :: Connection -> String -> [[DBValue]]
```

that performs an SQL query and returns a list of all rows in the result set *lazily*. The key idea is to delay the query for the actual rows until they are demanded by the program. The function `lazyResults` that takes a connection and a result handle and lazily returns a list of rows is implemented as follows:

```
lazyResults :: Connection -> ResultHandle -> [[DBValue]]
lazyResults (p,db) r
  | endOfResults (p,db) r = []
  | otherwise = send (NextRow db r row) &>
     (ensureNotFree row : lazyResults (p,db) r)
 where row free
```

Each row is queried on demand by sending a `NextRow` message and the result set is empty if `endOfResults` returns `True`. Note that we demand the contents of each row, when the corresponding constructor `(:)` of the result list is demanded. Since every call to `endOfResults` advances the result pointer to the next row, we may not be able to demand the contents of previous rows later.

Finally, the function `query` can be implemented using the `Query`-message and the function `lazyResults`:

```
query :: Connection -> String -> [[DBValue]]
query (p,db) sql
  | send (Query db sql resultHandle) p
  = lazyResults (p,db) (ensureNotFree resultHandle)
 where resultHandle free
```

Note that the presented functions are not i/o actions. Although i/o is performed by the Java application we communicate with, the communication itself is done with `send`-constraints outside the IO monad. Therefore, the presented functions can be considered unsafe and need to be used with care. Demand driven database access cannot be implemented without this kind of unsafe features. Recall that it is, e.g., not possible to implement a lazy `readFile` operation without unsafe features in the IO monad.

Note that the presented functions are *not* part of our database library. They are only used internally in the implementation of persistent predicates. The function `query` can be used to define a lazy version of `getDynamicSolutions` to retrieve all solutions of a persistent predicate abstraction on demand. Its implementation has to consider predicates that are combined from heterogeneous parts. Predicates stored in a database can be freely combined with others stored in files or those with facts held in main memory. The details are out of the scope of this paper. However, an interesting aspect of these details is how to convert between the datatype `DBValue` and Curry values automatically. An approach to this problem that employs type-oriented database specifications is discussed in Section 4. Beforehand, we consider an example that demonstrates lazy database access with persistent predicates:

```
main = do
  assert (foldr1 (<>) (map prime [3,5,7]))
  (p:ps) <- getDynamicSolutions prime
  print p
  retract (prime 5)
  print (head ps)
```

The output of this program is:

```
3
5
```

The first line inserts three prime numbers into the database. In the next line these prime numbers are queried from the database – at least conceptually. In fact, only the first prime is read from the database because it is demanded by the pattern `(p:ps)`. The next line prints the demanded prime on the screen. Then, the value 5 is deleted from the database. This does not affect the results that are not yet demanded because the database maintains a consistent snapshot of each result set. So in the next line, when the next prime is demanded and queried from the database, the value `5` is still an element of the result set that was queried before `5` was deleted. The value `7` is not queried from the database because it is not demanded by the program.

The communication performed by a call to `main` looks as follows: messages that are sent to the database server are aligned to the left and variable bindings that denote the response of the server are aligned to the right. Free variables are represented as terms `(VAR n)` during communication where `n` is an integer that identifies the variable.

```
Open "jdbc:mysql://localhost/currydb" (VAR 0)      (VAR 0) bound to 0
Update 0 "INSERT INTO primes VALUES (3),(5),(7)"
Query  0 "SELECT prime FROM primes" (VAR 0)        (VAR 0) bound to 0
EndOfResults 0 0 (VAR 0)                       (VAR 0) bound to False
NextRow 0 0 (VAR 0)                          (VAR 0) bound to [INT 3]
Update 0 "DELETE FROM primes WHERE prime = 5"
EndOfResults 0 0 (VAR 0)                       (VAR 0) bound to False
NextRow 0 0 (VAR 0)                          (VAR 0) bound to [INT 5]
```

SQL statements that do not return results are sent to the database in an `Update`-message. The `Open`- and the `Query`-message contain a free variable that is bound to a database- and a result-handle respectively. In this example both handles are equal to `0`. The `EndOfResults`-message is sent twice in the examples and also contains a free variable which is bound to `False` both times because the program does not demand all results. The free variable in the `NextRow`-message is bound to the list that represents the currently demanded prime each time the message is sent. Lazy access plays well together with transactions as the following slightly modified example demonstrates:

```
main' = do
  Just ps' <- transaction (do
    assert (foldr1 (<>) (map prime [3,5,7]))
    (p:ps) <- getDynamicSolutions prime
    print p
    retract (prime 5)
    return ps)
  print (head ps')
```

Here the modifications of the prime predicate are done inside a transaction and the second prime is demanded after the transaction has been committed. The output of the modified program is identical to the output of the original program, i.e., the value `5` is still queried from the database, although its retraction has been committed before it is demanded. We need to explicitly return the tail of the prime lists as result `ps'` of the transaction to be able to demand further primes. The variable `ps` is not visible outside the transaction. If the transaction was aborted, we could not demand further primes because the tail of the prime list would not be available. In case of an abortion `Nothing` would be the result of the transaction instead of `Just ps'`. The communication performed by a call to `main'` looks as follows:

```
Open "jdbc:mysql://localhost/currydb" (VAR 0)      (VAR 0) bound to 0
Update 0 "START TRANSACTION"
Update 0 "INSERT INTO primes VALUES (3),(5),(7)"
Query  0 "SELECT prime FROM primes" (VAR 0)        (VAR 0) bound to 0
EndOfResults 0 0 (VAR 0)                       (VAR 0) bound to False
NextRow 0 0 (VAR 0)                           (VAR 0) bound to [INT 3]
Update 0 "DELETE FROM primes WHERE prime = 5"
Update 0 "COMMIT"
EndOfResults 0 0 (VAR 0)                       (VAR 0) bound to False
NextRow 0 0 (VAR 0)                           (VAR 0) bound to [INT 5]
```

The interaction of lazy access with transaction management is handled by the database server. We do not have to take special precautions within our implementation. Especially, we do not explicitly store a consistent snapshot when we return the handle to the results of a query. We rather rely on the database system to return only results that are consistent with the point in time of the query. The Java program we communicate with merely defines wrapper functions for some database operations provided by the Java database interface (JDBC) and manages the different connections created by Curry programs. JDBC supports three different ways to query the results of a database query. All of them employ the notion of a cursor that navigates over the rows of a result set. The first only allows to retrieve the results subsequently one after the other, i.e., the cursor may only move forward step by step. The others support so called *scrollable* result sets where the cursor is allowed to move multiple steps at once forward and backward. The scrollable result sets differ in whether changes to the database that are made while the result handle is open are made visible or not. *Insensitive* result sets do not show such changes –

sensitive result sets do. Generally, scrollable result sets are less efficient than the one that supports only sequential access to the results. Therefore, we employ a non-scrollable result set in our implementation. We experienced this type of result set to be insensitive, which is crucial for lazy access. However, this may vary among different database servers. Hence, it may be necessary to use a scrollable, insensitive result set with other database systems. We believe that even an implementation that uses scrollable result sets will perform better than our original approach [4,5] based on parsing a string representation of the complete result set.

4 Type-Oriented Database Specifications

To implement the operations to access persistent predicates we need to convert between Curry values and values of type `DBValue`. Especially, to implement encapsulated search with the function `getDynamicSolutions`, we need to parse the rows of type `[DBValue]` that are returned by the function `query` introduced in Section 3.2. In this section we describe an approach to this problem using type-oriented database specifications. A similar technique has been employed in [8] to construct web user interfaces.

The special syntax to declare persistent predicates using the keyword `persistent` introduced in Section 2 is transformed into a call to a special function `persistent`n, where n is the arity of the declared predicate. Instead of prime numbers, we consider a slightly more complex example for a persistent predicate that stores information about persons:

```
data Name = Name String String
type YearOfBirth = Int

person :: Name -> YearOfBirth -> Dynamic
person persistent "db:currydb.persons"
```

A person has a name that consist of a first and a last name and the year of birth is stored to be able to compute the persons age. We use the given type signature and additional information (about the database driver) available to the run-time system to transform persistent predicate declarations when the program is loaded. The declaration of the persistent predicate `person` is internally transformed into

```
  person :: Name -> YearOfBirth -> Dynamic
  person = persistent2
             "jdbc:mysql://localhost/currydb persons"
             (cons2 Name (string "last") (string "first"))
             (int "born")
```

This declaration states that `person` is a persistent predicate with 2 arguments stored it the database `currydb` in a MySQL database on the local machine in the table `persons`. The table `persons` has 3 columns. The first two columns – *last* and *first* – store the name of a person, i.e., the first argument of `person`. The third column – *born* – stores the year of birth, i.e., the second argument of `person`. The specifications

```
cons2 Name (string "last") (string "first")
int "born"
```

resemble the structure of the argument types of `person`. The first resembles a constructor with two string arguments and the second represents an integer. The string arguments to the primitive specifications `string` and `int` denote the column names where the corresponding values are stored. Declarations like the one shown above are generated automatically from the provided type information. However, less intuitive column names would be selected by this automatic transformation.

The presented specifications serve different purposes. Firstly, they store information about the corresponding column names and SQL column types. Secondly, they store functions to convert between database and Curry values. We define a datatype `DBSpec` for these specifications as follows:

```
data DBSpec a = DBSpec [String] [String] (ReadDB a) (ShowDB a)
type ReadDB a = [DBValue] -> (a,[DBValue])
type ShowDB a = a -> [DBValue] -> [DBValue]
```

In fact, these types are a bit more complicated in the actual implementation. However, the presented types are sufficient for this description. The first two components of a `DBSpec` store the column names and types. A function of type `(ReadDB a)` is a parser that takes a list of database values and returns a value of type `a` along with the remaining unparsed database values. A function of type `(ShowDB a)` takes a value of type `a` and a list of database values and extends this list with the representation of the given value as database values. We can define the primitive combinator `int` presented above as follows:

```
int :: String -> DBSpec Int
int name = DBSpec [name] ["INT"] rd sh
 where
  rd (NULL   : xs) = (let x free in x, xs)
  rd (INT n  : xs) = (n, xs)

  sh n xs = (INT (ensureNotFree n) : xs)
```

The parser for integers reads one column from the list of database values and returns a free variable if it is a *null*-value. The show function extends the given database values with an integer value and suspends on free variables. Database specifications for other primitive types, viz. `string`, `float`, `bool` and `time`, can be defined similarly.

Complex datatypes can be represented by more than one column. Recall the specification for the name of a person introduced above:

```
cons2 Name (string "last") (string "first")
```

The combinator `cons2` can be defined as follows:

```
cons2 :: (a -> b -> c) -> DBSpec a -> DBSpec b -> DBSpec c
cons2 cons (DBSpec nsa tsa rda sha) (DBSpec nsb tsb rdb shb)
  = DBSpec (nsa++nsb) (tsa++tsb) rd sh
 where
  Cons a b = cons a b
  rd = rda />= \a -> rdb />= \b -> ret (Cons a b)
  sh (Cons a b) = sha a . shb b
```

This combinator takes a binary constructor `cons` as first argument. The subsequent arguments are database specifications corresponding to the argument types of the provided constructor. The name and type informations of the provided specifications are merged into the new specification, i.e., the arguments of the constructor are stored in subsequent columns of a database table. Finally, the read and show functions are constructed from the read and show-functions for the arguments. We use the predefined function composition `(.) :: (b->c) -> (a->b) -> (a->c)` to define the show function and monadic parser combinators for the read function:

```
(/>=) :: ReadDB a -> (a -> ReadDB b) -> ReadDB b
rda />= f = uncurry f . rda

ret :: a -> ReadDB a
ret a xs = (a,xs)
```

The function `uncurry :: (a->b->c) -> (a,b) -> c` transforms a binary function into a function on pairs.

The definition of the show function may be confusing at first glance: it matches a value `(Cons a b)` where `Cons` is a locally defined function equal to the constructor `cons` provided as first argument. Apart from constructors, in Curry also defined function symbols can be used in pattern declarations [2]. This allows us to define type-based combinators for arbitrary datatypes instead of only for specific ones. We provide similar combinators `cons1`, `cons3`, `cons4`, . . . for constructors of different arity.

The presented combinators allow for a concise declaration of database specifications that are used to convert between database and Curry values. The declarations are introduced automatically when a program is loaded. However, the programmer can also introduce them himself if he wants to control the column names, e.g., if he wants to access existing database tables. The generated converters are used internally to implement lazy access to persistent predicates based on the low level lazy database interface presented in Section 3.2.

The idea of type-oriented combinators seems to be applicable in a variety of applications. They bring the flavor of generic programming to a language without specific generic programming features. We plan to explore this connection in more detail in the future.

5 Conclusions

We described a lazy implementation of a functional logic database library for Curry. The library is based on persistent predicates which allow for transparent access to externally stored data, i.e., without storage specific code. We extend [4] with an implementation of lazy database access and simplified the declaration of transactions by discarding the function `transactionDB`.

We present an implementation of lazy database access that is both portable and maintainable because it is implemented in Curry using the concepts of ports [6] and not integrated into the run-time system using external functions. Moreover, our implementation supports a variety of database systems because it benefits from the extensive support for different database systems in Java. Using the ports-based lazy database interface, we implemented a low level lazy database interface for Curry. Based on this, we developed type-oriented converter specifications to implement a lazy version of `getDynamicSolutions` that encapsulates results of a `Dynamic` abstraction lazily. Values that are not demanded by the application program are not queried from the database in advance.

Although we do not introduce conceptual novelties concerning our database library, we demonstrate that quite technical implementation goals – viz. laziness, i.e., efficiency – can be achieved using high-level programming techniques. Functional logic programming is powerful enough to transparently and efficiently integrate database programming into its programming paradigm using functional logic programming techniques.

References

[1] Antoy, S., R. Echahed and M. Hanus, *A needed narrowing strategy*, Journal of the ACM **47** (2000), pp. 776–822.

[2] Antoy, S. and M. Hanus, *Declarative programming with function patterns*, in: *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)* (2005), pp. 6–22.

[3] Correas, J., J. 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)* (2004), pp. 104–119.

[4] Fischer, S., *A functional logic database library*, in: *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming* (2005), pp. 54–59.

[5] Fischer, S., "Functional Logic Programming with Databases," Master's thesis, Kiel University (2005), available at: `http://www.informatik.uni-kiel.de/~mh/lehre/diplom.html`.

[6] Hanus, M., *Distributed programming in a multi-paradigm declarative language*, in: *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)* (1999), pp. 376–395.

[7] Hanus, M., *Dynamic predicates in functional logic programs*, Journal of Functional and Logic Programming **2004** (2004).

[8] Hanus, M., *Type-oriented construction of web user interfaces*, in: *Proc. of the 8th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'06)* (2006), pp. 27–38.

[9] Leijen, D. and E. Meijer, *Domain specific embedded compilers*, in: *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99)* (1999), pp. 109–122.

[10] Peyton Jones, S., A. Gordon and S. Finne, *Concurrent Haskell*, in: *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL'96)* (1996), pp. 295–308.

[11] Wadler, P., *How to declare an imperative*, ACM Computing Surveys **29** (1997), pp. 240–263.