A Functional Logic Database Library

Sebastian Fischer

Christian-Albrechts-University of Kiel
Institute of Computer Science
Olshausenstr. 40
24098 Kiel, Germany
sebf@informatik.uni-kiel.de

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 describe a functional logic database library, based on persistent predicates, for Curry. Our library supports functional logic programming with databases in the background, i.e., the programmer can access a database without storage specific code employing functional logic programming techniques.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—*Multiparadigm Languages*

General Terms Languages

Keywords Curry, persistent storage, dynamic predicates, database library

1. Introduction

Programming languages need mechanisms to store data that persists among program executions. Internal data needs to be saved and recovered, or external data has to be represented and manipulated by an application. For instance, web applications often read data stored on the web 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.

1.1 Related Work

The notion of persistent predicates is introduced in [2] where a database and a file based 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 [2] has two major drawbacks: For database access it relies on side effects, and it does not support transactions.

Both problems are solved in [3] where database access is only possible inside the IO monad [5] and a transaction concept is provided. Eliminating side effects is essential in the context of functional logic programming languages which are based on sophisticated evaluation strategies [1]. A transaction concept is mandatory to support web based applications that simultaneously handle different requests. We extend the library presented in [3] by a database implementation of persistent predicates.

A combinator library for Haskell, which is used to construct database queries with relational algebra, is provided by [4]. 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 [4] *syntactically* integrates a domain specific language into the Haskell programming language, persistent predicates *transparently* integrate database access into an existing 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.

2.1 Basic Operations

The basic operations for persistent predicates stored in a database involve assertion, retraction and query. 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

WCFLP'05 September 29, 2005, Tallinn, Estonia.

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

and

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

are provided. 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 the return type of `retract` is `IO ()` unlike presented in [3] and that all facts that equal the value to retract are deleted, if there is more than one such fact. The original return type of `retract` presented in [3] is `IO Bool`, and its behavior was changed to reflect the `DELETE` statement present in SQL. All matching records are removed by this statement, and to determine whether an entry was deleted, the resulting table has to be compared to the original, which is a needless inefficiency. To identify whether a fact exists in the database, the programmer can use the function `isKnown`, hence, he is not reliant on the function `retract` to return a boolean value.

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

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

takes a dynamic predicate 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, and

```
isKnown :: Dynamic -> IO Bool
```

detects whether a given fact exists in the database. Note that `isKnown` can be implemented as

```
isKnown d =
 getDynamicSolution (const d) >>= return . isJust
```

The function `getKnowledge` provided by [3] is not supported in combination with persistent predicates stored in a database. We discuss this limitation in Section 3.

2.2 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)
transactionDB :: String -> IO a -> IO (Maybe a)
abortTransaction :: IO a
```

The function `transaction` is used to start transactions that use persistent predicates not stored in a database but in files. To start a transaction employing database predicates, the function `transactionDB` is supplied with a database name and an IO action to perform. Both functions perform the given IO action as a transaction and wrap its result in the `Maybe` type if the transaction completes normally or return `Nothing` if it fails or is aborted with `abortTransaction`.

Calls to `transactionDB` can be nested, to perform a transaction in different databases. For instance, if an IO action `t` involves predicates from two databases `database1` and `database2`, the call

```
transactionDB "database1"
 (transactionDB "database2" t >>=
  maybe abortTransaction return)
```

performs the IO action `t` as a transaction in both given databases. The function `maybe` is defined as

```
maybe :: a -> (b -> a) -> Maybe b -> a
maybe x _ Nothing = x
maybe _ f (Just x) = f x
```

and useful for programming with optional values.

2.3 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 persistent predicates to more complex predicates, we provide two different forms of conjunction. One combines two values of type `Dynamic`, the other combines a `Dynamic` value 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                                    (1)
 |> x+2 == y
```

resembles the SQL query

```
SELECT tab1.col, tab2.col
  FROM primes AS tab1, primes AS tab2                  (2)
 WHERE tab1.col + 2 = tab2.col
```

2.4 Database Specific Combinators

Unfortunately, this database query cannot be constructed at run time, because then information about the structure of `x+2 == y` and the columns in which `x` and `y` are stored is not available. The database specific combinators presented in this section serve the first purpose: They record the structure of the conditions in data terms available at run time. Columns, however, have to be referenced explicitly with these combinators. Although a programmer can use them, it is slightly inconvenient and not recommended. Therefore, we provide a program transformation that automatically associates variables with database table columns by generating expressions built from the combinators presented in this section. Hence, the programmer does not need to take the database tables into account, but he is free to do so and use database specific code. If he does, of course, he cannot change the internal storage mechanism of the database predicates.

An abstract data type `SQLExp a` represents expressions used in the `WHERE` part of an SQL query. Its constructors are hidden, so only provided combinators can be employed to construct such expressions. The combinator

```
(.|>) :: Dynamic -> SQLExp Bool -> Dynamic
```

is used to combine a database predicate and an expression of type `SQLExp Bool`. The function

```
val :: a -> SQLExp a
```

is used to lift ground values to the type `SQLExp a` and

```
col :: Int -> SQLExp a
```

is used to reference columns in database tables. The columns of all involved tables are numbered from zero and columns from combined predicates are consecutively numbered.

We provide various operators to construct expressions of type `SQLExp a`. For example, comparison operators like

```
(.==) :: SQLExp a -> SQLExp a -> SQLExp Bool
(.<=) :: SQLExp Int -> SQLExp Int -> SQLExp Bool
```

and arithmetic operators like

```
(.+) :: SQLExp Int -> SQLExp Int -> SQLExp Int
(.*) :: SQLExp Int -> SQLExp Int -> SQLExp Int
```

are lifted to the `SQLExp` type. Many others, such as boolean conjunction and disjunction, are available; a complete list is too large for the scope of this description.

The `Dynamic` abstraction (1) can be expressed using database specific combinators as follows:

```
\(x,y) ->
 prime x <> prime y                                    (3)
 .|> col 0 .+ val 2 .== col 1
```

Our program transformation automatically transforms abstraction (1) into the database specific version (3) which can be translated at run time into the SQL query (2).

Note that the provided combinators are type safe. They are implemented using a *phantom type*: The type parameter of the polymorphic type `SQLExp a` is not used in the definition of this type but only to encode the typing rules for the combinators used to construct expressions of type `SQLExp a`. Constructing an ill-typed expression like `val 42 .== val ""` is prevented from the Curry type checker; hence, the combinators not only ensure syntactically correct but also type safe SQL queries. Since only provided combinators can be used to construct the expressions, the type signatures of these combinators ensure type safety of the constructed values. Unfortunately, there is one combinator that potentially destroys type safety since it constructs expressions of arbitrary type: The function `col :: Int -> SQLExp a` is used to reference table columns of any column type. The programmer using this combinator has to ensure that he references columns containing values with correct types, since otherwise, type errors in the `WHERE` part of a database query cannot be prevented. The translation of values of type `SQLExp Bool` into the `WHERE` part of an SQL query is straightforward, since the provided combinators resemble the most common operations available in SQL.

2.5 Representing Arguments in a Database Table

The queries that are generated for a database predicate are also determined by the representation of its arguments in the corresponding database table. Primitive values such as numbers or strings can be stored in one column, and queries can restrict these columns according to the values. The persistent predicate

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

is stored in the database `currydb` in a table `primes` with one column `col` which is automatically generated with the SQL statement

```
CREATE TABLE primes (col INT)
```

if it does not exist while loading the program. The conditional predicate

```
primePair :: (Int,Int) -> Dynamic
primePair (x,y) = prime x <> prime y |> x+2 == y
```

describes a prime pair employing a condition attached to the dynamic predicate using `(|>)`. An SQL query describing a prime pair employs the condition

```
tab1.col + 2 = tab2.col
```

in its `WHERE`-part since the argument `x` of the predicate `prime` is stored in the column `col` of the table referenced as `tab1` and `y` in the column `col` of the table referenced as `tab2`. All arguments of database predicates could be stored in a single column of a table since there are functions `readTerm` and `showTerm` converting arbitrary values into strings and vice versa.

Type of Argument	Representation
primitive types	single column
optional values	single column, *null* for `Nothing`
record types	multiple columns
lists	separate table, *null* for the empty list
everything else	string representation in a single column

Table 1. Representation of Arguments

More sophisticated storage mechanisms, however, give rise to more detailed database queries. Especially optional values, record types and lists can be handled differently to allow for efficient translation of restrictions into database queries. Table 1 shows how different Curry types are represented in a database. The subsequent sections describe the storage of optional values, record types and lists.

2.5.1 Optional Values and Record Types

In database tables usually the null value is used to represent missing values. In Curry the type `Maybe a` serves the same purpose; hence, values of type `Maybe a` are represented in one column by the string representation of the value of type `a` or as null value if they equal `Nothing`. This approach enables the programmer to access existing database tables with columns that can contain null values via the `Maybe` type. This is an advantage since the notion of optional values represented by null values is transferred to the Curry program where optional values are represented as values of type `Maybe a`.

Record types are types similar to tuples, i.e., non-recursive types with a single constructor. To independently restrict parts of a record, these parts need to be stored in separate columns. For instance, we could define the predicate `primePair` to store its facts directly in a database table:

```
primePair :: (Int,Int) -> Dynamic
primePair persistent "db:currydb.primePairs"
```

If the argument of `primePair` were stored in a single column in a table created by

```
CREATE TABLE primePairs (pp TEXT)
```

the elements of a prime pair could not be restricted independently in an SQL query. If instead a table with two columns is created

```
CREATE TABLE primePairs (fst INT, snd INT)
```

a dynamic abstraction describing a prime triple

```
\(x,y,z) -> primePair (x,y) <> primePair (y,z)
```

can be translated into the SQL query

```
SELECT tab1.fst, tab1.snd, tab2.snd
  FROM primePairs AS tab1, primePairs AS tab2
 WHERE tab1.snd = tab2.fst
```

The presented database library stores records in multiple columns per default to enable more flexible conditions on parts of a record.

2.5.2 Lists

Because of their prominent role in functional programming, lists are not stored as strings but considered separately. Consider a predicate storing lists of prime numbers:

```
primeList :: [Int] -> Dynamic
primeList persistent "db:currydb.primeLists"
```

The arguments of `primeList` are stored in a separate table instead of being converted into a string. Each element of the list can be stored in a single row of the separate table, and a reference to the list elements is stored in the original table. To preserve the order of the list, an additional index is stored along with each entry. A null value is stored instead of the reference, to represent the empty list. Hence, the predicate `primeList` is stored in *two* tables created by

```
CREATE TABLE primeLists (ref INT)
CREATE TABLE elems (ref INT, idx INT, elem INT)
```

This representation of lists can greatly enhance the performance of database queries if restrictions on list elements are translated into SQL. For instance, consider the dynamic abstraction

```
\ps -> primeList ps |> elem 7 ps
```

which describes all prime lists that contain the number 7. Assuming a lot of stored prime lists with very few of them containing 7, the corresponding query would be very expensive if the list elements were not available to an SQL query. Every stored list would have to be queried and tested afterwards. Storing each list element in an own row of a separate table gives rise to a more sophisticated query: The references of lists containing the number 7 can be queried in advance using the SQL query

```
SELECT ref FROM elems WHERE elem = 7
```

Then, these references can be used to query only those lists that satisfy the given condition.

2.6 Interfacing Existing Databases

The presented database library allows for transparent storage of algebraic data types in a relational database. It can, however, also handle existing database tables using database predicates. If a table `primePair` created by

```
CREATE TABLE primePair (fst INT, snd INT)
```

is present in the database `currydb`, the persistent predicate declaration

```
primePair :: Int -> Int -> Dynamic                (4)
primePair persistent "db:currydb.primePair"
```

can be used to access the data stored in that table. This predicate declaration can be automatically generated, and the programmer can use it as provided, or he can change it to structure the arguments of the database predicate. The predicate defined by

```
primePair :: (Int,Int) -> Dynamic
primePair persistent "db:currydb.primePair"
```

can also be used to access the database table `primePair`.

To generate persistent predicate declarations for existing relational database tables, the functions

```
interface :: String -> IO ()
interfaceTables :: String -> [String] -> IO ()
```

are provided which both take a database name as argument and generate a Curry file with declarations of dynamic predicates interfacing tables of the given database. The function `interface` generates predicates for all tables in the database, and `interfaceTables` is provided with a list of table names to interface with. For instance, the call

```
interfaceTables "currydb" ["primePair"]
```

writes declaration (4) of the predicate `primePair` into a file called `currydb.curry`.

To generate a predicate declaration for a given database table, we need to compute a type signature for the predicate that resembles the column types of the given table. Each column is represented by exactly one argument of the predicate and the argument types are chosen to reflect the type of the column they represent. The column types provided by SQL are mapped to the Curry types `Bool`, `Int`, `Float`, `Char` and `String` appropriately. We also provide a data type `SQLDate` to support a special column type representing dates in SQL. Its constructor is hidden to prevent pattern matching. Instead, functions are provided to construct, decompose or compare values of type `SQLDate`:

```
sqlDate :: Int -> Int -> Int -> SQLDate

year :: SQLDate -> Int
month :: SQLDate -> Int
day :: SQLDate -> Int

before :: SQLDate -> SQLDate -> Bool
```

All those functions can be translated into database queries using our program transformation since there are equivalent versions lifted to the `SQLExp` data type.

3. Simulating Dynamic Knowledge

The interface of the dynamic predicate library presented in [3] provides a function `getKnowledge` which is used to conceptually retrieve all currently known information. This knowledge is returned as a function converting values of type `Dynamic` into values of type `Success`, and thus, it enables the programmer to apply a logic programming style to dynamic predicates. Since `getKnowledge` is not supported in combination with persistent predicates stored in a database, the programmer is forced to use the alternative conjunction combinators `(<>)` and `(|>)` which also allow for a logic programming style but cannot be applied as flexible as the concurrent constraint conjunction `(&)`. Especially, recursive dynamic predicates cannot be defined using `(<>)`. Hence, `getKnowledge` seems to be more flexible in combination with `(&)` and `getAllSolutions` than `(<>)` and `(|>)` in combination with `getDynamicSolutions`. This section, however, shows that the programming style enabled by `getKnowledge` can be imitated using `getDynamicSolutions` and, thus, can be applied to persistent predicates stored in databases as well.

To demonstrate the mentioned programming style, consider an example program defining a recursive predicate on lists of prime numbers:

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

primePair :: (Int,Int) -> Dynamic
primePair (x,y) = prime x <> prime y |> x+2 == y

primePairList::(Dynamic->Success)->[Int]->Success
primePairList known [x,y]
  = known (primePair (x,y))

primePairList known (x:y:zs)
  = known (primePair (x,y))
  & primePairList known (y:zs)
```

```
main = do
  known <- getKnowledge

  getAllSolutions (primePairList known)
    >>= mapIO_ print
```

If `prime` were a persistent predicate stored in a database, calling `getKnowledge` would not be allowed. The programmer can, however, call `getDynamicSolutions` instead to retrieve the information needed to define the function `known` explicitly. Using a datatype `Dyn` representing the facts of the predicate `primePair`, `primePairList` can be defined similar to the example above:

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

primePair :: (Int,Int) -> Dynamic
primePair (x,y) = prime x <> prime y |> x+2 == y

data Dyn = PrimePair (Int,Int)

primePairList :: (Dyn->Success)->[Int]->Success
primePairList known [x,y]
  = known (PrimePair (x,y))

primePairList known (x:y:zs)
  = known (PrimePair (x,y))
  & primePairList known (y:zs)

main = do
  primePairs <- getDynamicSolutions primePair

  let known (PrimePair p)
    = p =:= foldr1 (?) primePairs

  getAllSolutions (primePairList known)
    >>= mapIO_ print
```

The explicit definition of the function `known` is still possible in more complex examples, although the programmer has to control what data is retrieved in advance. The similarities of the given examples encourage to believe that the presented transformation could be done automatically. The information on restrictions, however, needs to be transferred from the applications of `known` to its definition to avoid querying too much information from a potentially very large database. Future implementations of this database library should consider to provide such a transformation. Experience has to reveal whether it is useful or too inefficient in practice.

4. Empirical Results

To document the usefulness of the presented approach, we provide a prototype implementation and compare its performance to that of the existing file based implementation. Four different queries are presented to highlight advantages and deficiencies of one approach compared to the other. Neither the file based nor the database implementation can prevail over the other implementation in all situations.

As a basis for our tests we employ a database of publications that we stored employing both persistent predicates that use files for external storage and others that employ a relational database. We cannot store the whole database with the file based implementation, since all facts are held in memory at run time. Hence, we access only 20,000 publications with file based persistent predicates while 100,000 publications are stored with database predicates. This appears to be unfair competition; however, research has been made for decades how to efficiently access relational databases, and database predicates presumably will be used with large amounts of data that does not fit in main memory. So, this imbalance is justified with regard to practical aspects. In Section 4.2 we discuss how the different amounts of stored data and different counts of returned results affect the response time of both implementations.

The presented measurements are average times taken on an AMD Athlon(tm) XP 3200+ with 1 GB main memory. We performed ten queries and eliminated outliers to compute an average from the remaining values. The file based implementation needs about 10 seconds to read the stored 510MB into main memory. In applications that perform few database queries like CGI scripts this is not negligible; nevertheless, we discard this load time for the presented measurements.

4.1 Queries

To query the database, we employ a minimal data type representing publications:

```
data Publication = Publication Id Title [Id]

type Id = Int
type Title = String
```

A publication stores an identifier, a title and a list of identifiers referencing other publications. The database stores many other attributes, but this interface suffices for the presented queries.

The first query simply queries one publication with a specific identifier.

```
intMatch title
  = publication (Publication 10000 title refs)
 where
  refs free
```

It can be used to measure the time the implementation needs to search through all entries. Whether a given entry has a specific identifier can be tested very fast.

The second query is similar to the first; however, it has a more complex condition, so both implementations will take more time to solve it.

```
stringMatch title
  = publication (Publication ref title refs)
  |> "Curry" `substringOf` title
 where
  ref,refs free
```

All publications that contain the word `"Curry"` in their title are requested. The function `substringOf` is defined as

```
substringOf :: String -> String -> Bool
substringOf s [] = null s
substringOf s (c:cs)
  = startsWith (c:cs) s || substringOf s cs

startsWith :: String -> String -> Bool
startsWith _ [] = True
startsWith (c:cs) (p:ps)
  = p==c && startsWith cs ps
```

and there is a corresponding function

```
substringOf'
 :: SQLExp String -> SQLExp String -> SQLExp Bool
```

to efficiently translate it into an SQL query.

The third query differs from the second only in the word searched for in the title.

```
stringMatch' title
  = publication (Publication ref title refs)
  |> "Database" `substringOf` title
 where
  ref,refs free
```

Since the database contains much more publications about databases than about Curry, this query has a lot more results. Hence, we can compare the run time of this query with that of the previous one to detect what time is needed to read the results of a query compared to the time that is needed to compute this result.

Finally, we consider a query incorporating a restriction on the stored list of references.

```
elemMatch title
  = publication (Publication ref title refs)
  |> 3264 `elem` refs
 where
  ref,refs free
```

This query relies on the database implementation to restrict the queried values *in advance* by the condition on the publication's references. Otherwise, every entry would have to be read from the database to test the condition.

4.2 Results

Table 2 shows the results of our measurements, listing for each query the count of results and the time required to retrieve them. The first query is answered by both implementations in less than a

Predicate	**Files** : **(20,000)**		**Database** : **(100,000)**	
	count :	seconds	count :	seconds
`intMatch`	1 :	0.7	1 :	0.4
`stringMatch`	0 :	11.7	5 :	1.3
`stringMatch'`	367 :	11.2	1758 :	15.3
`elemMatch`	1 :	1.3	4 :	0.8

Table 2. Results of the Performance Comparison

second. For significant results a larger amount of data is required; the file based implementation, however, cannot handle arbitrary amounts of data, so we cannot increase the significance of the result for this query.

The second and third query expose differences between the two implementations: While the file based implementation answers both queries equally fast, the response time of the database implementation highly depends on the count of requested results. Therefore, we can conclude that the time required to read the results is a crucial factor for the response time and that the answers are computed quite fast. This observation is supported by the fast response to the query with few results compared to the file based approach. A more sophisticated implementation of low-level database access potentially increases its performance substantially. The current implementation employs a command line database interface, which is accessed via standard IO.

Since the count of requested results determines the response time of the database implementation, the last query shows that the restriction of lists significantly improves the response time. If the condition on references were not considered in the database query, all 100,000 publications would have to be queried and tested afterwards; this would definitely inhibit a response in less than a second. The fast response time of the file based implementation shows that the condition on references can be tested faster than the substring condition of the second and third query.

5. Conclusions

We described 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 [3] with a database implementation and modify the given interface in three aspects:

- The type of the function `retract` changes from `Dynamic -> IO Bool` to `Dynamic -> IO ()`,
- the function `transactionDB` is included to support transactions with database predicates, and
- the function `getKnowledge` is not supported in combination with database predicates.

We describe an intermediate library with database specific combinators to express conditions on arguments of persistent predicates that can be directly translated into an SQL query. These database specific combinators allow for a more efficient database access. As database specific combinators destroy the transparency of persistent predicates, we provide a program transformation to automatically introduce them. Thus, programs can be written without database specific code and later be transformed to enable more efficient database access. We also give a general scheme to represent arguments of dynamic predicates in a database table. Optional values of type `Maybe a` make use of null values, and can be used to access existing tables that employ null values. Records are stored in multiple columns to enable more flexible restrictions in the generated SQL queries. For the same reason, lists are stored in a separate table which greatly enhances the performance of some queries. We also provide a mechanism to interface with existing databases. Persistent predicate declarations can be automatically generated and may be adapted to provide additional structure to the argument types.

For future work we plan

- a program transformation that automatically translates calls to `transaction` into calls to `transactionDB` if database predicates are involved,
- a program transformation based on the ideas given in Section 3 to support the function `getKnowledge` in combination with database predicates, and
- a different implementation of low-level database access to enhance the moderate performance in processing large result sets.

We have compared a prototype implementation of the presented approach to the existing file based implementation with encouraging results. Although database access can be improved to reduce the response time for large result sets, the presented implementation successfully competes with the file based implementation. It is mandatory when the stored data is too voluminous to be held in main memory.

References

[1] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[2] 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)*, pages 104–119. Springer LNCS 3057, 2004.

[3] M. Hanus. Dynamic predicates in functional logic programs. *Journal of Functional and Logic Programming*, 2004(5), 2004.

[4] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99)*, pages 109–122. ACM SIGPLAN Notices 35(1), 1999.

[5] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.