Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Functional Logic Programming with Databases

Sebastian Fischer

April 19, 2005

Institute of Computer Science and Applied Mathematics
Programming Languages and Compiler Construction

Supervised by:
Prof. Dr. Michael Hanus
Bernd Brassel

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angebenen Hilfsmittel verwendet habe.

Kiel,

_____

Abstract

Programmers need mechanisms to store application specific data that persists multiple program runs. To accomplish this task, programmers 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 provide an implementation of persistent predicates, based on relational databases, for the functional logic programming language Curry. Our library supports functional logic programming with databases in the background, i.e., the programmer can access a database employing functional logic programming techniques, and not database related.

We provide a prototype implementation of the presented library to document the usefulness of our approach and have compared it to the existing file based implementation with encouraging results.

Acknowledgments

I wholeheartedly thank Lena for her love and for her patience during the preparation of this thesis.

This thesis is part of my degree at the Christian-Albrechts-University of Kiel. I was working on this topic in the group for Programming Languages and Compiler Construction from October 2004 to April 2005.

My sincere thanks go to my supervisors Michael Hanus and Bernd Brassel for many fruitful discussions and constructive comments during the development of the work presented in this thesis. I thank Michael Hanus for his detailed and challenging remarks on a first draft of this thesis. Bernd Brassel made valuable suggestions for form and content for which I am deeply grateful. Thank you for your patience in repeatedly proof-reading the draft versions of this thesis!

I also appreciate many provoking questions by Frank Huch at a seminar talk on the presented work.

Finally, I thank my parents for their encouraging support and enduring trust.

Contents

1. Persistent Storage

Programming languages need mechanisms to store data that persist 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. Often interactive forms, which allow for the manipulation of the internal data via a web based interface, are provided.

The most common approach to persistent storage is to use files to store the data that shall persist among program executions. Data can be stored in an application specific format, e.g, the Java programming language offers a mechanism called *serialization* to write objects to streams of data. Thus, in Java programs this mechanism could be employed to store data in files. To make the stored information available to other applications possibly written in different programming languages, the data format has to be language independent. The eXtensible Mark-up Language (XML) serves that purpose. Many programming languages provide libraries to read and write XML data, and sometimes they also provide a mechanism to automatically convert XML data into a language specific format. This mechanism is called XML language binding, and it allows for transparent access to XML data; i.e., the programmer does not need to be aware of XML if he uses XML language binding.

Access to data stored in files becomes inefficient for very large files. Usually Databases are used to efficiently access a large amount of stored data. Relational database systems based on the relational data model presented by [2] dominate the database world. In a relational database data is stored in tables that can be divided into rows and columns. For instance, scientific publications could be stored in a table, each record in an own row; and the table could include columns for the name of the author, title of the publication, etc.

The Structured Query Language (SQL) [13] is used to access and maintain data stored in a relational database. It provides statements to create and delete databases and tables as well as statements to access these tables, e.g., query information. A query *selects* some rows of a table using a condition and *projects* some columns to be queried. For instance, the SQL statement

```
SELECT title FROM publications WHERE author="Euclid"
```

retrieves all titles of publications written by Euclid, since the column *title* of the table *publications* is projected, and all rows where the column *author* contains the value `"Euclid"` are selected. Relational database systems provide tools to interactively enter SQL queries and view the results. This kind of access, however, becomes inefficient if a large amount of queries has to be sent, or the queries results are voluminous and require

further processing. Thus, application programs can interface with database systems to run queries and process their results.

1.1. Database Access

Database access is handled differently in common programming languages. In Java JDBC[1] can be used to access ODBC[2]-compliant databases. It enables the programmer to send SQL queries to a database and process the results wrapped as Java objects. This kind of access is error-prone, since SQL queries are represented as strings in a Java program, and thus, their syntax is not checked by the compiler. An approach to transparently store objects of the programming language Python in a relational database is described in [11]. The programmer can store and retrieve Python objects in a relational database without dealing with SQL statements. HaSQL[3] and HaskellDB [10] are approaches to database access in Haskell. The latter provides combinators to construct database queries using relational algebra that are automatically translated into SQL queries. Thus, it ensures the queries to be syntactically correct and type-safe. Logic programming provides a natural approach to relational databases, since logic programming languages have a built-in notion of predicates. The notion of persistent predicates, which provide transparent database access in Prolog, is introduced in [3]. The definition of persistent predicates is stored externally, e.g., in a relational database, and thus, persists over multiple program runs and is available to other programs. An application using persistent predicates does not rely on a specific storage mechanism which therefore can be exchanged without touching the code of the application; [3] provides implementations based on both files and relational databases.

An implementation of dynamic predicates that adheres to the evaluation strategy of Curry [1] is presented in [7]. Predicates are called dynamic if their definition changes at runtime and persistent if they are dynamic and externally stored to persist multiple program runs. The library presented in [7] implements persistent predicates by storing facts in files. This thesis adds a database implementation to the library, and thus, it allows for functional logic programming with databases. Internal data of a Curry application can be transparently stored in a relational database, and an interface to an existent database can be generated automatically. Functional logic programming combines the best of the two main directions of declarative programming: Computing with partial information and nondeterministic search for solutions from logic programming is combined with high level abstraction mechanisms from functional programming, i.e., higher order functions, lazy evaluation and algebraic data types. The notion of predicates offers a natural interface to relational databases, and higher order functions are very useful to construct complex combined dynamic predicates. This work integrates database access into a functional logic programming paradigm which is an advantage over the most existing database libraries where the programmer has to learn a database

[1] http://java.sun.com/products/jdbc/
[2] Open DataBase Connectivity
[3] http://members.tripod.com/~sproot/hasql.htm

query language or needs to be familiar with relational algebra to construct database queries.

This section mentioned mechanisms to persistently store data used in application programs. This data can be stored in files or relational databases, and persistent predicates allow for transparent storage of data in programming languages that support predicates and binding of free variables. This thesis presents a functional logic database library for the programming language Curry based on persistent predicates, and a prototype implementation is used to document the usefulness of the approach.

The next chapter introduces Curry and SQL. The presented library is introduced in Chapter 3. In Chapter 4 we discuss implementation details before we compare the new implementation with the previously presented file based approach in Chapter 5. In Chapter 6 we consider related and future work and Chapter 7 concludes.

2. Preliminaries

2.1. Curry

Curry [8] is a multi-paradigm programming language combining functional, logic and concurrent programming. In this section we give an introduction to Curry, and we introduce an example program we will refer to in future sections.

2.1.1. Operational Semantics

The integration of functional and logic programming is reflected by the operational semantics of Curry which is based on lazy evaluation combined with a possible instantiation of free variables. On ground terms the operational model is similar to lazy functional programming, while free variables are nondeterministically instantiated like in logic languages; hence, different results can be computed for different variable instantiations. Logic programming focuses on the bindings of free variables while in functional programming one is interested in the computed result. Since Curry is an integrated functional logic programming language, an *answer expression* is a pair of variable bindings and a computed result. Because more than one result may be computed nondeterministically, initial expressions are reduced to disjunctions of answer expressions. A *disjunctive expression* is a multi-set of answer expressions usually written as

```
{x_1=v_1,...,x_m=v_m} e_1 | ... | {x_1=v_1',...,x_m=v_m'} e_n
```

where the bindings of free variables are enclosed in curly brackets preceding the computed result, and the alternatives are separated by vertical bars. For instance, the rules

```
f 0 = 2
f 1 = 3
```

define a function `f` which yields `2` if applied to `0` and `3` if applied to `1`. If `x` is a free variable, the call `f x` reduces to the disjunctive expression

```
{x=0} 2 | {x=1} 3
```

since `x` can be bound to `0` to reduce `f 0` to `2`, or `x` can be bound to `1` to reduce `f 1` to `3`. If they are clear from the context the bindings of free variables are omitted in disjunctive expressions.

In Curry, nested expressions are evaluated lazily, i.e. the leftmost outermost function call is selected for reduction in a computation step. If in a reduction step an argument value is a free variable and demanded by an argument position of the left-hand side of some rules, it is either instantiated to the demanded values nondeterministically or

the function call suspends until the argument is bound by another concurrent computation. Binding free variables is called *narrowing*; suspending calls on free variables is called *residuation*. Curry supports both models since it aims at providing a platform for different declarative programming styles, and which strategy is "right" depends on the intended meaning of the called function.

2.1.2. Data Type and Function Declarations

Curry supports algebraic data types that can be defined by the keyword `data` followed by a list of constructor declarations divided by `|`. The declaration

```
data List a = [] | a : List a
```

defines the ubiquitous type of lists usually written as `[a]` with the constructors `[]` for the empty list and `(:)` which takes an element and a list and constructs a list with the given element as head and the given list as tail. Instead of `1:2:3:[]` one can write `[1,2,3]` for convenience. As an example of a function operating on lists, consider list concatenation:

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

As strings are represented as lists of characters in Curry, this function can also be used to concatenate strings. To model optional values, Curry defines the type `Maybe a` as

```
data Maybe a = Nothing | Just a
```

So a value of type `Maybe a` is either `Nothing` or `Just x` for some value `x`. Type synonyms can be defined by the keyword `type`. The declarations

```
data Publication = Publication Id [Author] Title [Id]
data Author = Author Name (Maybe Address)

type Id = Int
type Title = String
type Name = String
type Address = String
```

model a data type representing publications. A publication consists of a numeric identifier, a list of authors, a title and a list of identifiers that represent other publications referenced by the publication. An author consists of a name and an optional address.

In Curry functions are defined by rules with an optional condition; function application is written in prefix notation except for infix operators which can be used in infix notation or in prefix notation if enclosed in brackets. Functions can be defined using pattern matching by writing constructor applications as arguments. For instance, the function `title` that gets the title of a publication can be defined as:

```
title :: Publication -> Title
title (Publication _ t _ _) = t
```

The underscore is an anonymous variable and used for values that are not required for the definition of a function. The functions

```
identifier :: Publication -> Id
authors :: Publication -> [Author]
references :: Publication -> [Id]
```

and

```
name :: Author -> Name
address :: Author -> Maybe Address
```

can be defined analogously. In Curry, anonymous functions can be defined using lambda abstractions. The abstraction

```
(\p -> identifier p == 42)
```

is equivalent to the function

```
idIs42 p = identifier p == 42
```

but no name is assigned to the anonymous function. Such functions are useful in combination with higher order functions like `filter`

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs
```

Functions can be arguments and results of other functions, and a function need not be applied to all arguments. As an example consider the function `publicationById` that takes a list of publications and an identifier and returns the publication with the given identifier if there is such a publication:

```
publicationById :: [Publication] -> Id -> Publication
publicationById ps n = head (filter (\p -> n == identifier p) ps)
```

This declaration employs a lambda abstraction to select the appropriate publication. The function fails if there is no matching publication, since the function `head`, which returns the first element of a non-empty list, fails if applied to the empty list. Instead of the lambda abstraction

```
(\p -> n == identifier p)
```

function composition

```
(.) :: (a -> b) -> (c -> a) -> c -> b
f . g = \x -> f (g x)
```

can be used to identify the publication with the identifier `n`:

```
(n==) . identifier
```

The expression `(n==)` is a partial application of `(==)` to `n` and thus takes another argument which is compared to the value of `n`.

As another example of a higher order function consider the definition of `foldr`

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

and the application

```
foldr (\x y -> x++", "++y) "" ["Euclid","Hilbert"].
```

The function `foldr` is applied to a lambda abstraction as first argument that puts a comma between its two arguments, the empty string as second argument and the list of two mathematicians as third argument. The application evaluates to the string `"Euclid, Hilbert, "`. All given mathematicians are listed in a single string separated by commas. Unfortunately, there is also a comma after the last name, since the empty string given as second argument of `foldr` is also separated from the resulting string by a comma. The function `foldr1` is a variant of `foldr` that is only defined for non-empty lists and does not require an identity element. The application

```
foldr1 (\x y -> x++", "++y) ["Euclid","Hilbert"]
```

evaluates to the string `"Euclid, Hilbert"`.

For a clear separation of imperative and declarative parts of a program, Curry supports monadic IO introduced by [12]. An IO action that returns a value of type `a` has the type `IO a` and can be created using

```
return :: a -> IO a
```

Multiple IO actions can be sequentially combined by

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

For example

```
getChar >>= putChar
```

is a combined IO action copying one character from stdin to stdout.

Since Curry is a functional *logic* language it provides a notion of free variables that can be bound by unification, nondeterministic search for solutions and nondeterministic operations. Predicates are expressed as functions with result type `Success` - a type that has only a single value `success` which is used as result for satisfiable constraints. In Curry, nondeterministic operations can be defined by overlapping rules. The function `(?)` defined as

```
(?) :: a -> a -> a
x ? _ = x
_ ? x = x
```

nondeterministically returns one of its arguments, hence,

```
member :: [a] -> a
member = foldr1 (?)
```

is an operation nondeterministically evaluating to an element of the given list. The `member` function can be used to define a predicate `publication` that is satisfied if the given publication can be unified with a publication in a given list of publications.

```
publication :: [Publication] -> Publication -> Success
publication = (=:=) . member
```

Indeed this definition without the type signature would work for lists of arbitrary types. We declare a specialized type since we want to define the predicate only on publications. The constraint equality operator `(=:=)` binds free variables and has result-type `Success`, thus, it fails if its arguments cannot be unified. The predicate `publication` nondeterministically unifies its second argument with an element of its first argument. If more than one list element can be unified with the second argument, more than one result is computed; and if no element of the first argument matches the second, the operation fails. We can now define a function that finds the title of a publication with a given identifier in a list of publications in (at least) two alternative ways: We can employ a purely functional approach

```
titleById :: [Publication] -> Id -> Title
titleById ps = title . publicationById ps
```

or we can use the predicate `publication`:

```
titleById ps n | publication ps (Publication n X t X) = t
 where
  t free

X = let x free in x
```

To avoid the declaration of unused free variables, we define a singleton free variable `X`. Think of it as an underscore; the occurrences of `X` in the definition of `titleById` denote *different* free variables. We use a local declaration introduced by `let` *local decls* `in` *exp* to define the singleton variable. We only need to declare `t` as free variable in the local declarations introduced with the keyword `where` in the declaration of `titleById`. Beyond free variables, functions can be defined inside local declarations introduced by `let` or `where`.

The first declaration is very short, but it employs rather special functions `title` and `publicationById`. The predicate `publication` is more general, since it could as well be used to define the functions `authorsById`, `refsByTitle`, etc. The search goal can be augmented with boolean conditions, which resemble a filter in a purely functional approach:

```
idByKeyword :: [Publication] -> String -> Id
idByKeyword ps keyword
  | publication ps (Publication n X t X)
  & (keyword `substringOf` t) =:= True
    = n
 where
  n,t free
```

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
```

The IO action

```
getAllSolutions :: (a -> Success) -> IO [a]
```

computes all results for a predicate and is used to encapsulate nondeterminism. So another approach to compute identifiers of publications that have a given keyword in their title is to define a predicate holding for such identifiers and use `getAllSolutions` to compute all solutions for the predicate:

```
idHasKeyword :: [Publication] -> String -> Id -> Success
idHasKeyword ps keyword n
  = publication (Publication n t X X)
  & (keyword `substringOf` t) =:= True
 where
  t free
```

Now the IO action

```
getAllSolutions (idHasKeyword ps keyword)
```

computes all identifiers for publications with titles that contain the given keyword.

We saw that predicates provide a convenient access to a list of stored publications. But we do not need to store data in lists, we can as well use predicates to define facts representing stored data. For instance, a predicate storing (some) prime numbers can be defined as

```
prime :: Int -> Success
prime = (=:=) . member [2,3,5,7,11,13,17]
```

employing the presented `member` operation and a list of prime numbers. We can also directly define a predicate identifying prime numbers:

```
prime' :: Int -> Success
prime' 2 = success
prime' 3 = success
prime' 5 = success
prime' 7 = success
prime' 11 = success
prime' 13 = success
prime' 17 = success
```

In this definition each fact is given explicitly in the program source. Both predicate declarations are equivalent, i.e., they can be used interchangeably.

In this section we introduced the functional logic programming language Curry and saw that a predicate can be versatilely employed to query a given list of publications. The next section introduces SQL which is used to query relational databases.

2.2. Structured Query Language

The Structured Query Language (SQL) [13] is a standardized language to query and manipulate relational databases. The relational data model [2] presents two kinds of languages to access a relational database: relational algebra and relational calculus. Relational algebra uses algebraic combinators like *join* and *projection* to combine relations represented by tables. SQL is an exponent of relational calculus languages where the desired results are described in a way easily accessible to humans. In this section we introduce those SQL statements necessary to follow the examples presented in this thesis and present the data types for table columns supported by SQL.

2.2.1. SQL Statements

SQL supports so called *schema*-statements used to maintain a relational database, *data*-statements to access tables in a database and *transaction*-statements to perform several statements as an atomic unit. As an example for a schema-statement consider the `CREATE TABLE` statement

```
CREATE TABLE authors (name TEXT,address TEXT)
```

which is used to create tables in a database. The example statement creates a table called *authors* with two columns *name* and *address*. The columns have been associated with types; there are types for numbers, characters, strings or even dates. Usually SQL keywords are written in `CAPITAL` letters while table and column names are not. Another SQL-schema-statement is `DESCRIBE table` which can be used to query information about a database table. The statement

```
DESCRIBE authors
```

requests the names of the table's columns, their types and some other information concerning *null* values, keys and default values. Consider the first two columns of the answer, specifying the column's names and their types:

```
Field   Type   Null   Key    Default Extra
name    TEXT   YES           NULL
address TEXT   YES           NULL
```

The values are those we used in the `CREATE TABLE` statement above. SQL-data-statements involve insertion and deletion

```
INSERT INTO authors VALUES ("Euclid","Alexandria")

DELETE FROM authors WHERE name="Euclid"
```

as well as selection

```
SELECT address FROM authors WHERE name="Euclid"
```

of table rows. With the `INSERT`-statement the specified values are inserted as new row of the given table. In the example only one data record is inserted, multiple records can be inserted separated by commas. Rows can be deleted by specifying a condition restricting the rows to delete. The `SELECT`-statement contains a similar restriction to specify the rows to select from the given table. Additionally, the columns that shall be selected are given. As result of the `SELECT` statement a table with one row and one column is returned

```
Alexandria
```

since there is only a single record for Euclid in the database, and the column *address* was projected in the query. As a default, all SQL-statements are instantly committed and therefore change the database immediately. To temporarily disable *auto-commit* mode, the *transaction*-statement

```
START TRANSACTION
```

is provided. All statements that change the database and are performed after the transaction starts have to be committed explicitly by

```
COMMIT
```

or can be discarded using

```
ROLLBACK
```

For instance, the statements

```
START TRANSACTION
INSERT INTO authors VALUES ("Euclid","Alexandria")
ROLLBACK
```

do not affect the database; no additional data record is inserted into the table *authors*. For the SQL-transaction-statements to take effect, the database has to be set up to support a transactional table engine if it does not support one by default.

2.2.2. Column Types

Values can be stored in various formats in the columns of database tables. The presented column types can be divided into three categories: numeric types, date and time types and string types.

There are numeric types for integer values and for floating point numbers. The presented types differ in their range, and some are simply synonyms for others. Table 2.1 shows integer types and their signed and unsigned ranges. Table 2.2 shows different

column type	signed min	signed max	unsigned max	synonyms
TINYINT	-128	127	255	BIT,BOOL,BOOLEAN
SMALLINT	-32768	32767	65535	
MEDIUMINT	-8388608	8388607	16777215	
INT	-2147483648	2147483647	4294967295	INTEGER

Table 2.1.: Integer Column Types

column type	absolute min	absolute max	synonyms
FLOAT	$\approx 1.2 \cdot 10^{-38}$	$\approx 3.4 \cdot 10^{38}$	
DOUBLE	$\approx 2.2 \cdot 10^{-308}$	$\approx 1.8 \cdot 10^{308}$	REAL
DECIMAL	$\approx 2.2 \cdot 10^{-308}$	$\approx 1.8 \cdot 10^{308}$	DEC,NUMERIC,FIXED

Table 2.2.: Floating Point Column Types

floating point column types. The minimal and maximal absolute values are given; if the types are specified UNSIGNED negative values are not allowed. Values of type DECIMAL are stored as string with one character for each digit.

SQL supports data types for date and time values. Table 2.3 shows three of them

column type	format
DATE	YYYY-MM-DD
TIME	HH:MM:SS
DATETIME	YYYY-MM-DD HH:MM:SS
TIMESTAMP	YYYY-MM-DD HH:MM:SS

Table 2.3.: Date and Time Column Types

and their format. A TIMESTAMP column is useful for recording the date and time of operations changing the table contents. It is automatically set to the current date and time if no value or the *null* value is assigned.

Strings can be stored in binary format or as characters with an associated character set. Table 2.4 shows different column types to store strings and binary large objects (*blobs*). The types BINARY and VARBINARY as well as the BLOB types store binary strings rather than non-binary strings. Non-binary strings have an associated character set and are compared case insensitive. Columns of type VARCHAR and VARBINARY as well

column type	maximum length
CHAR	255
VARCHAR	255
BINARY	255
VARBINARY	255
TINYTEXT	255
TINYBLOB	255
TEXT	65,535
BLOB	65,535
MEDIUMTEXT	16,777,215
MEDIUMBLOB	16,777,215
LONGTEXT	4,294,967,295
LONGBLOB	4,294,967,295

Table 2.4.: String Column Types

as columns with `TEXT` or `BLOB` types require only storage for actual stored characters, while columns of type `CHAR` and `BINARY` require a fixed amount of storage regardless of their contents. Their are additional column types for strings and sets of strings. Columns of type `ENUM(value_1,...,value_n)` store values chosen from the given list of values, while the *null* value and a special error value `''` is also allowed. The values are represented internally as integers. Columns of type `SET(value_1,...,value_n)` store up to 64 values chosen from the given list. `SET` values are represented internally as integers.

3. A Functional Logic Database Library

In this chapter we consider an implementation of persistent predicates in Curry employing a relational database for external storage of the predicates facts. The use of a relational database allows for handling large amounts of data that cannot be held in memory to apply the file based approach. Moreover it enables conditions on dynamic predicates to be solved by the database system, not by the run-time system.

Using persistent predicates for database access, hides the database from the programmer and allows for a functional logic programming style. Programming with predicates is well known to logic programmers, and features of functional programming like higher order functions and partial evaluation can be applied to combine persistent predicates. As seen in Section 2.1, predicates enable a versatile access to stored ground values. The familiar programming paradigm is a key advantage of the presented database library.

The facts of a persistent predicate can be transparently stored as rows of a database table associated with the predicate, i.e., the programmer does not need to be aware of the storage mechanism. Each argument of the predicate can be stored in an own column, and complex arguments like tuples can be stored in multiple columns to allow for queries restricting only parts of it. Since lists are frequently used in Curry, they are considered separately; in Section 3.2.3 we describe how lists are stored in database tables.

In the remainder of this chapter we describe the interface of the presented library, discuss how arguments of persistent predicates are stored in database tables and present a set of database specific combinators that can be directly translated into database queries.

3.1. Interface of the Library

In this section we describe the interface of the presented database library. Indeed it is similar to the interface presented in [7] but adds database specific combinators which can be directly translated into database queries.

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 values of type `Publication` in the table `publications` in the database `currydb`:

```
publication :: Publication -> Dynamic
publication persistent "db:currydb.publications"
```

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 database must

be accessible via the employed back-end on the computer that runs the program. The result type of persistent predicates is `Dynamic` which is conceptually similar to `Success` the result type of ordinary predicates. 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.

3.1.1. 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 *authors* created by

```
CREATE TABLE authors (name TEXT, address TEXT)
```

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

```
authors :: String -> String -> Dynamic
authors persistent "db:currydb.authors"
```

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

```
data Author = Author Name (Maybe Address)

type Name = String
type Address = String

author :: Author -> Dynamic
author persistent "db:currydb.authors"
```

can also be used to access the database table *authors*. Section 3.2 discusses how arguments of persistent predicates can be represented as columns of a database table.

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.

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. To formalize the translation,

we define a mapping τ translating SQL column types into Curry types:

$$\begin{aligned}
Bools &:= \{\texttt{BIT}, \texttt{BOOL}, \texttt{BOOLEAN}\} \\
Ints &:= \{\texttt{TINYINT}, \texttt{SMALLINT}, \texttt{MEDIUMINT}, \texttt{INT}, \texttt{INTEGER}, \texttt{BIGINT}\} \\
Floats &:= \{\texttt{FLOAT}, \texttt{DOUBLE}, \texttt{REAL}, \texttt{DECIMAL}, \texttt{DEC}, \texttt{NUMERIC}\} \\
Strings &:= \{\texttt{VARCHAR}, \texttt{TINYTEXT}, \texttt{MEDIUMTEXT}, \texttt{TEXT}, \texttt{LONGTEXT}\} \\
SQLTypes &:= Bools \cup Ints \cup Floats \cup \{\texttt{CHAR}\} \cup \{\texttt{DATE}\} \cup Strings \\
CurryTypes &\supseteq \{\texttt{Bool}, \texttt{Int}, \texttt{Float}, \texttt{Char}, \texttt{SQLDate}, \texttt{String}\}
\end{aligned}$$

$$\tau : SQLTypes \to CurryTypes, \quad t \mapsto \begin{cases} \texttt{Bool} & , t \in Bools \\ \texttt{Int} & , t \in Ints \\ \texttt{Float} & , t \in Floats \\ \texttt{Char} & , t = \texttt{CHAR} \\ \texttt{SQLDate} & , t = \texttt{DATE} \\ \texttt{String} & , t \in Strings \end{cases}$$

Arguments of type `Bool`, `Int`, `Float`, `Char` and `SQLDate` can be generated from corresponding SQL column types. The type `SQLDate` is only defined to represent dates in SQL format and described in detail in Section 3.1.5.

To generate the above declaration of the persistent predicate `authors`, the function `interfaceTables` can be called with the database's and table's names. The call

```
interfaceTables "currydb" ["authors"]
```

writes the above declaration of the predicate `authors` into a file called `currydb.curry`.

3.1.2. Basic Operations

The basic operations for persistent predicates stored in a database involve assertion, retraction and query. Because the definition of dynamic predicates changes over time, their access is only possible inside the IO Monad [12] to provide an explicit order of evaluation. To manipulate the facts of a dynamic predicate, the functions

```
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 the values of the arguments are known. As an example recall the database predicate `author` interfacing the table *authors*. The IO action

```
assert (author (Author "Euclid" (Just "Alexandria")))
```

inserts a row with the appropriate values into the table *authors*, and the IO action

```
retract (author (Author "Euclid" (Just "Alexandria")))
```

deletes all such rows from the table *authors*. Note that the return type of `retract` is `IO ()` unlike presented in [7] 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 [7] 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 dynamic predicate can have multiple solutions computed nondeterministically. 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`. For instance, the query

```
getDynamicSolutions (\name -> author (Author name (Just "Alexandria")))
```

computes a list of all names of authors from Alexandria that are stored in the table *authors*. 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 [7] is not supported in combination with persistent predicates stored in a database. See Section 5.1 for a discussion of this limitation.

3.1.3. 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)
transactionDB :: String -> IO a -> IO (Maybe a)
abortTransaction :: IO a
```

The function `transaction` is used to start transactions that do not use persistent predicates stored in a database. 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`.

To perform a transaction in different databases, calls to `transactionDB` can be nested. 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.

3.1.4. Combining Dynamic Predicates

Often information needs to be queried from more than one dynamic predicate at once, or a query has to be restricted with a boolean condition. For instance, a predicate storing authors of publications can be combined with a predicate storing publications along with the name of its author and its title. We simplify the example presented in Section 2.1 to illustrate a combination of dynamic predicates:

```
data Publication = Publication Name Title
data Author = Author Name (Maybe Address)

type Name = String
type Title = String
type Address = String

publication :: Publication -> Dynamic
publication persistent "db:currydb.publications"

author :: Author -> Dynamic
author persistent "db:currydb.authors"
```

The example defines two persistent predicates stored in a database *currydb*. One stores publications by the name of the author and the title of the publication. The other one was already established in Section 3.1.2 and stores names of authors along with an optional address. To query the titles of publications written by authors from Alexandria, we need to combine both predicates. Moreover, to query the names of authors of publications with titles that contain the word “Geometry”, we need to combine the predicate `publication` with a boolean condition expressing this property.

Dynamic predicates can be combined to more complex predicates using 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
```

As an example of a combination of two different dynamic predicates consider a query for titles of publications written in Alexandria. It can be expressed in Curry by

```
writtenInAlexandria :: Title -> Dynamic
writtenInAlexandria title
  = publication (Publication name title) <>
    author (Author name (Just "Alexandria"))
 where
  name free
```

and the IO action

```
getDynamicSolutions writtenInAlexandria
```

returns a list of all titles of publications written by authors from Alexandria currently stored in the tables *publications* and *authors*. A query for names of authors that worked on geometry can be expressed by

```
workedOnGeometry :: Name -> Dynamic
workedOnGeometry name
  = publication (Publication name title) |>
    "Geometry" `substringOf` title
 where
  title free
```

and the IO action

```
getDynamicSolutions workedOnGeometry
```

returns the list of all authors of publications with a title containing the word "Geometry" currently stored in the table *publications*.

The presented combinators can be employed to construct `Dynamic`-abstractions that resemble database queries. The functions `writtenInAlexandria` and `workedOnGeometry` contain everything necessary to construct a database query. To illustrate this correspondence they are given as lambda abstractions along with an SQL statement resembling the abstraction.

The function `writtenInAlexandria` is equivalent to the lambda abstraction

```
\title ->
  publication (Publication name title) <>
  author (Author name (Just "Alexandria"))
```

The declaration of the free variable `name` has been omitted to emphasize the resemblance to the database query

```
SELECT publications.title
  FROM publications, authors
 WHERE publications.name = authors.name
   AND authors.address = "Alexandria"
```

The pattern variable `title` of the abstraction corresponds to the projected column *title* in the database query, and the tables *publications* and *authors* used in the query resemble the corresponding predicates used in the body of the lambda abstraction. Moreover, the variable `name` is shared among the two predicate calls. This restriction is expressed in the first part of the `WHERE`-clause of the database query. The second part of the `WHERE`-clause expresses the restriction posed by the ground value in the second argument position of the constructor `Author`.

The function `workedOnGeometry` is equivalent to the lambda abstraction

```
\name ->
  publication (Publication name title) |>
  "Geometry" `substringOf` title
```

The declaration of the free variable `title` has been omitted similar to the previous example. The abstraction corresponds to the database query

```
SELECT publications.name
  FROM publications
 WHERE publications.title REGEXP "Geometry"
```

The projected column *name* resembles the pattern variable `name`, and the predicate `publication` is represented by the corresponding table in the database query. The substring condition is translated into a restriction, expressed by a simple regular expression. The regular expresison is just the string "Geometry" that has to occur somewhere in the value of the column *title* in the table *publications* to satisfy the restriction.

Unfortunately, we cannot construct the conditions presented in the previous examples at runtime since the structure of the corresponding expression is not available to the runtime system, and the used variables are not associated with the corresponding columns of the tables: For instance, the internal structure of

```
"Geometry" `substringOf` title
```

is not available at runtime and the *name*-columns of the tables *publications* and *authors* have to be associated with the shared variable `name` to compute the restriction

```
publications.name = authors.name
```

Both problems are addressed in later sections: Section 3.3 introduces combinators that can be directly translated into SQL queries and Section 4.1 presents a program transformation automatically augmenting a program with such combinators.

3.1.5. Special Purpose Combinators

SQL has built-in data types for text or sequences of characters, numerical values like integers and floats, and there is also a special data type representing dates. In Section 3.1.1

we showed how the column types of SQL are represented in Curry. The only column type not represented by a primitive Curry type is `DATE` which is modeled by the abstract data type `SQLDate`. 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 the program transformation described in Section 4.1. For instance, if the data type for publications is augmented with a date of publication

```
data Publication = Publication Name Title SQLDate
```

the query function

```
writtenBeforeChrist :: Title -> Success
writtenBeforeChrist title
  = publication (Publication X title date) |>
    date `before` sqlDate 0 1 1
```

would describe all titles of publications written before Christian era. The equivalent database query is

```
SELECT publications.title
  FROM publications
 WHERE publications.date < "0000-01-01"
```

There are other notable predicates that can be translated into database queries. We already used the function `substringOf` to query publications about geometry. The functions `startsWith` and `endsWith` are equally useful, can be translated similarly and are therefore provided by the database library:

```
startsWith :: String -> String -> Bool
endsWith :: String -> String -> Bool
substringOf :: String -> String -> Bool
```

Those functions can be used as infix operators and are translated into database queries using regular expressions.

In this section we discussed the interface of the presented database library. The key operations involving assertion, retraction and query of dynamic predicates were presented as well as two different conjunction operators to build more complex dynamic predicates. Finally, we documented the resemblance of complex dynamic predicate abstractions to database queries and motivated the database specific combinators presented in Section 3.3.

3.2. Arguments of Predicates in a Database Table

The representation of a persistent predicate's arguments in a database table determines the queries generated for this predicate. 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

```
authors :: String -> String -> Dynamic
authors persistent "db:currydb.authors"
```

introduced in Section 3.1.1 is stored in the database `currydb` in a table *authors* with two columns which is automatically generated with the SQL statement

```
CREATE TABLE authors (name TEXT, address TEXT)
```

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

```
authors name "Alexandria" |> name=="Euclid"
```

describes the author Euclid of Alexandria employing both a condition attached to the dynamic predicate using `(|>)` and a ground value in the second argument position of `authors`. An SQL query describing Euclid of Alexandria needs the condition

```
name="Euclid" AND address="Alexandria"
```

in its `WHERE`-part since the first argument of the predicate `authors` is stored in the column *name* and the second in the column *address*. All arguments could be stored like this in a single column of a table since there are functions `readTerm` and `showTerm` converting arbitrary values into strings and vice versa. More sophisticated storage mechanisms,

Type of Argument	**Representation**
primitive: `Int, Float, Char, String`	single column
record types	multiple columns, maybe separate table
lists	separate table, *null* for empty list
optional values	single column, *null* for `Nothing`
everything else	string representation in single column

Table 3.1.: Representation of Arguments

however, give rise to more detailed database queries. Especially record types and lists can be handled differently to allow for efficient translation of restrictions into database queries. Table 3.1 shows how different Curry types can be represented in a database. The subsequent sections describe the storage of optional values, record types and lists.

Similar to Section 3.1.1, we formalize the translation of Curry types into SQL column types before we give examples in the following sections. A Curry type is mapped into a word over SQL column types, each representing one database table. A Curry type can be associated to more than one column type if it is a record type; and if it embodies a list, it is stored in multiple tables.

Let $CurryTypes$ be a set with the following properties:

$$\begin{aligned} CurryTypes &\supseteq \{\texttt{Bool}, \texttt{Int}, \texttt{Float}, \texttt{Char}, \texttt{Date}, \texttt{String}\} \\ \forall n \in \mathbb{N}\, \forall t_1, \ldots, t_n \in CurryTypes &: record(t_1, \ldots, t_n) \in CurryTypes \\ \forall t \in CurryTypes &: optional(t) \in CurryTypes \\ \forall t \in CurryTypes &: list(t) \in CurryTypes \end{aligned}$$

and

$$SQLTypes \supseteq \{\texttt{BOOL}, \texttt{INT}, \texttt{FLOAT}, \texttt{CHAR}, \texttt{DATE}, \texttt{TEXT}\}$$

For instance, the type `Publication` defined as

```
data Publication = Publication Author Title
data Author = Author Name (Maybe Address)

type Title = String
type Name = String
type Address = String
```

is represented by

$$record(record(\texttt{String}, optional(\texttt{String})), \texttt{String}).$$

We define a mapping ζ as:

$$\zeta : CurryTypes \to SQLTypes^+$$

$$t \mapsto \begin{cases} \texttt{BOOL} & , t = \texttt{Bool} \\ \texttt{INT} & , t = \texttt{Int} \\ \texttt{FLOAT} & , t = \texttt{Float} \\ \texttt{CHAR} & , t = \texttt{Char} \\ \texttt{DATE} & , t = \texttt{SQLDate} \\ \zeta(t_1) \cdot \ldots \cdot \zeta(t_n) & , \exists n \in \mathbb{N}\, \exists t_1, \ldots, t_n \in CurryTypes : t = record(t_1, \ldots, t_n) \\ \zeta(t') & , \exists t' \in CurryTypes : t = optional(t') \wedge |\zeta(t')| = 1 \\ \texttt{TEXT} & , \exists t' \in CurryTypes : t = optional(t') \wedge |\zeta(t')| > 1 \\ \texttt{INT} & , \exists t' \in CurryTypes : t = list(t') \\ \texttt{TEXT} & , \text{otherwise} \end{cases}$$

Concatenation on words and a length function are defined as usual:

$$\begin{aligned} (x_1, \ldots, x_m) \cdot (y_1, \ldots, y_n) &:= (x_1, \ldots, x_m, y_1, \ldots, y_n) \\ |(x_1, \ldots, x_n)| &:= n \end{aligned}$$

Record types are mapped to multiple columns that store the components of the record; see Section 3.2.1 for a complete discussion. Optional values of type $optional(t')$ are stored in *one* column, since a single *null* value is used to represent a missing value. If a value of type t' is usually stored in multiple columns, a string representation of this

value is stored in one column instead. Note that the value of $\zeta(list(t'))$ is `INT` because for lists of type $list(t')$ a separate table with column types $\zeta(t')$ is created and the list elements are referenced by an `INT`-reference (cf. Section 3.2.3). Other Curry types, i.e., those with multiple constructors like

```
data Either a b = Left a | Right b
```

are represented by a single column of type `TEXT`; in Section 3.2.2 we discuss an alternative approach.

3.2.1. 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` can be 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`.

In this section we employ the `Maybe` type to store an optional address in a record type representing authors of a publication.

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, recall the predicate `author`

```
data Author = Author Name (Maybe Address)

author :: Author -> Dynamic
author persistent "db:currydb.authors"
```

introduced in Section 3.1.1. If the argument of `author` was stored in a single column in a table created by

```
CREATE TABLE authors (author TEXT)
```

the predicate describing Euclid of Alexandria can be expressed in Curry by

```
author name (Just "Alexandria") |> name=="Euclid"
```

An equivalent database query needs to restrict the single column with the condition

```
author='Author "Euclid" (Just "Alexandria")'
```

since the value of type `Author` is saved as string in the single column of the table *authors*. If instead a table with two columns is created

```
CREATE TABLE authors (name TEXT, address TEXT)
```

a database query describing Euclid of Alexandria can be formulated using the condition

```
name="Euclid" AND address="Alexandria"
```

in the WHERE-part of the query. This representation of records allows for more flexible restrictions, since columns need not always be restricted with ground values. Recall the query introduced in Section 3.1.4

```
SELECT publications.title
  FROM publications, authors
 WHERE publications.name = authors.name
   AND authors.address = "Alexandria"
```

The condition

```
publications.name = authors.name
```

can only be expressed if the name of an author is saved in a separate column of the table *authors*. The presented database library stores records in multiple columns per default to enable more flexible conditions on parts of a record. To complete the section on storing records we describe two alternative approaches for storing the columns of records. The database library implements the one described first, the other one is primarily considered to prepare the reader for the next section where we present how to store lists in separate tables.

To describe the first approach, we consider an example involving the record types Publication and Author:

```
data Publication = Publication Author Title
data Author = Author Name (Maybe Address)

type Title = String
type Name = String
type Address = String

publication :: Publication -> Dynamic
publication persistent "db:currydb.publication"
```

The parts of the records can be stored in multiple columns of the table associated with the database predicate. Therefore, in the given example the author's name and address can be stored in two columns of the table *publications* created by

```
CREATE TABLE publications (name TEXT, address TEXT, title TEXT)
```

To insert the book "Elements" by Euclid of Alexandria into the table *publications*, the IO action

```
assert (publication
  (Publication (Author "Euclid" (Just "Alexandria")) "Elements"))
```

has to be performed. Table 3.2 shows the table *publications* after this assertion. The presented database library stores records like presented in this paragraph. There is an alternative approach which can employ sharing to reduce the memory requirements of the stored records and is described in the remainder of this section.

name	address	title
Euclid	Alexandria	Elements

Table 3.2.: Table *publications* in database *currydb*

The columns representing the part of a record can be stored in a separate table with an additional reference identifying the entries of that table. To store the facts of the predicate `publication` defined above, two tables can be created with the SQL statements

```
CREATE TABLE publications (author INT, title TEXT)
CREATE TABLE authors (ref INT, name TEXT, address TEXT)
```

The Tables 3.3 and 3.4 show the created tables after the assertion

```
assert (publication
  (Publication (Author "Euclid" (Just "Alexandria")) "Elements"))
```

author	title
1	Elements

Table 3.3.: Table *publications* in database *currydb*

ref	name	address
1	Euclid	Alexandria

Table 3.4.: Table *authors* in database *currydb*

The record `Author "Euclid" (Just "Alexandria")` is stored in a separate table *authors*, and the reference identifying this record is inserted into column *author* of the table *publications*. If many publications written by few authors have to be stored, this mechanism allows to share the record for an author by using its reference multiple times in the table *publications*, and thus, the authors do not have to be stored again with every of their publications in the database. References could be counted to prevent the retraction of a still referenced record.

In this section we showed how records are represented in the columns of a database table. The parts of the record are stored in own columns; hence, they can be restricted independently from each other. An alternative approach that employs a separate table for a stored record was presented. This approach can be adapted to store lists which is described in the next section.

3.2.2. Variant Records and Recursive Types

Variant records are non-recursive data types with multiple constructors. As an example consider an alternative data type representing publications:

```
data Publication
  = Book Author Title
  | Article Author Title Journal
  | PhdThesis Author Title School
  | InProceedings Author Title Proceedings

data Author = Author Name (Maybe Address)
data Journal = Journal Name Publisher Volume Number
data School = School Name Address
data Proceedings = Proceedings Title Publisher

type Title = String
type Name = String
type Address = String
type Publisher = String
type Volume = Int
type Number = Int
```

We did not yet address how to store variant records except for optional values of type `Maybe a`; and we also did not address recursive data types[1]. Variant records can not be stored like records for two reasons:

- The alternative values may require a different number of columns of different types to store their parts.
- The table does not store information about which value is stored.

In the example, values constructed with `Article` require 7 columns to store their parts, while values constructed with `PhdThesis` or `InProceedings` require only 5 columns. The constructor `Book` is stored in 3 columns. We cannot store values of type `Publication` in a single table, unless we used different columns for every alternative, placing *null* values in the unused columns on an assertion. Padding a table with *null* values squanders storage space, since columns have to be reserved which are rarely used. Storing the parts of different alternatives in the same columns would inhibit reconstruction, unless information is stored, which alternative to use to rebuild the stored value. For instance, storing values constructed with `PhdThesis` and `InProceedings` in the same columns would be possible, since both require the same column types. For the same reason we cannot reconstruct a value, since we do not know which constructor to use. We can store information about the constructor in a separate column; but in general we cannot prevent using some *null* values in each inserted table row.

In Section 3.2.1 we showed how to store record types in a separate table. This approach can be adapted to store variant records and recursive data types: Each alternative can be stored in a separate table, so different columns can be reserved for each table, and no *null* values need to be placed in unused columns. The values stored in the separate

[1] Lists will be considered separately in the next section.

table are referenced like presented in Section 3.2.1; but in the referencing table an index identifying the constructor must be stored along with the reference to determine which table is referenced. In the example five tables can be created to store publications

```
CREATE TABLE publications (publication INT, alternative INT)
CREATE TABLE books (ref INT, name TEXT, address TEXT, title TEXT)
CREATE TABLE articles ...
CREATE TABLE phds ...
CREATE TABLE inprocs ...
```

and the values inserted by

```
assert (publication
 (Book (Author "Euclid" (Just "Alexandria")) "Elements"))
```

are shown in Tables 3.5 and 3.6. Since `Book` is the first constructor in the definition of

publication	alternative
1	0

Table 3.5.: Table *publications* in Database *currydb*

ref	name	address	title
1	Euclid	Alexandria	Elements

Table 3.6.: Table *books* in Database *currydb*

`Publication`, the value `0` is stored in the column *alternative*. Hence, the reference `1` identifies the value representing Euclid's Elements stored in the table *books*.

3.2.3. Lists

Because of their prominent role in functional programming, lists are not stored as strings but considered separately. Recall the original data type `Publication` introduced in Section 2.1:

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

Every publication has a unique identifier, and, instead of only one, multiple authors can be stored in a list. Another list stores identifiers of cited publications. The list of authors is stored in a separate table similar to the approach presented at the end of Section 3.2.1. Instead of storing one author in the separate table, all authors are inserted with the same reference. To preserve the order of the list, an additional index is stored along with the entry. A similar table is employed to store the list of citations; hence, the predicate

```
publication :: Publication -> Dynamic
publication persistent "db:currydb.publications"
```

is stored in three tables created by the SQL statements

```
CREATE TABLE publications
  (id INT, authors INT, title TEXT, citations INT)
CREATE TABLE authors (ref INT, idx INT, name TEXT, address TEXT)
CREATE TABLE citations (ref INT, idx INT, id INT)
```

The IO action

```
assert (publication (Publication 17
  [Author "Euclid" (Just "Alexandria")] "Elements" [42,7,11])
```

asserts the "Elements" written by Euclid of Alexandria. The tables 3.7, 3.8 and 3.9 show the values inserted by this assertion. The table *authors* is similar to the one

id	authors	title	citations
17	1	Elements	1

Table 3.7.: Table *publications* in database *currydb*

ref	idx	name	address
1	0	Euclid	Alexandria

Table 3.8.: Table *authors* in database *currydb*

ref	idx	id
1	0	42
1	1	7
1	2	11

Table 3.9.: Table *citations* in database *currydb*

used in Section 3.2.1, but it also stores the indices of the stored authors to preserve their order. Table *citations* stores the citations `[42,7,11]`. Similar to the assertion of missing optional values, a *null* value is inserted instead of a reference to the elements of the list if an empty list is to be stored.

We presented an approach to store arguments of persistent predicates in columns of database tables. For record arguments we discussed two different approaches storing multiple columns in the table associated with the predicate or in a separate table. An approach to store variant records and arbitrary recursive data types was discussed but is not considered further, since it is not realized in the current implementation. Since they are frequently used by functional programmers, lists and optional values are addressed separately.

3.3. Database Specific Combinators

In Section 3.1 we showed that dynamic predicate abstractions resemble database queries. These, however, can not be constructed at runtime since information about the structure

of the conditions and the columns in which variables are stored is missing at runtime. The database specific combinators presented in this section serve the first purpose: They record the structure of the conditions in data terms available at runtime. Columns, however, have to be referenced explicitly with these combinators, so while using them as programmer is possible it is slightly inconvenient and not recommended. The program transformation presented in Section 4.1 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.

Recall the predicate `workedOnGeometry` introduced in Section 3.1.4:

```
workedOnGeometry :: Name -> Dynamic
workedOnGeometry name
  = publication (Publication name title) |>
    "Geometry" `substringOf` title
 where
  title free
```

The predicate's condition `"Geometry" `substringOf` title` can be translated into an equivalent expression of the type `SQLExp Bool`:

```
substringOf' (val "Geometry") (col 1)
```

accounting that the variable `title` is stored in the second column of the table storing facts for the persistent predicate `publication`. The abstract data type `SQLExp a` represents expressions used in the `WHERE`-part of an SQL query. Its constructors are hidden, so only the provided combinators can be employed to construct these 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.

Various operators to construct values of type `SQLExp a` are provided. In Section 3.1.4 a query for titles of publications written in Alexandria was introduced to demonstrate the combination of dynamic predicates:

```
\title ->
  publication (Publication name title) <>
  author (Author name (Just "Alexandria"))
```

This query was recognized to be equivalent to the database query

```
SELECT publications.title
  FROM publications, authors
 WHERE publications.name = authors.name
   AND authors.address = "Alexandria".
```

To express the `WHERE`-part of this query, an expression of type `SQLExp Bool` is required that represents a boolean conjunction of equalities.

To construct such expressions, comparison operators like

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

and boolean operators like

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

are provided. Many others, such as combinators to construct arithmetic expressions, are available; see Appendix A for a complete list.

Employing these combinators, the `WHERE`-part of the query above can be expressed in Curry by

```
col 0 .== col 2 .&& col 3 .== val "Alexandria".
```

If this expression is attached to the dynamic predicate, an efficient database query can be generated to get the solutions of the predicate. This example is reviewed in Section 4.1.6 which clarifies how to compute the numbers of the columns.

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 expressions. 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 these combinators 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.

An SQL query not only includes a restriction but also projections on some database table's columns. For instance, the above query

```
SELECT publications.title
  FROM publications, authors
 WHERE publications.name = authors.name
   AND authors.address = "Alexandria"
```

selects only the *title* column of the table *publications* instead of all columns of all involved tables. To express projections in Curry, there is another combinator

```
(.!!) :: Dynamic -> [Int] -> Dynamic
```

which takes a list of column numbers as argument identifying the columns that have to be projected. The query shown above is automatically generated from the Curry predicate

```
\title ->
  publication (Publication name title) <>
  author (Author name (Just "Alexandria")) .|>
  col 0 .== col 2 .&& col 3 .== val "Alexandria" .!! [1]
```

The programmer is encouraged to use the presented combinators only if he is certain about the column numbers. Referencing columns numerically is error-prone and restricting or projecting the wrong columns certainly results in unexpected program behavior.

In this chapter we presented a functional logic database library based on persistent predicates stored in a relational database. After we described its interface, we pointed out the resemblance of predicate abstractions to database queries and discussed how argument values of persistent predicates can be represented as columns of a database table. Based upon this, we presented database specific combinators which express restrictions and projections on such columns and can be directly translated into SQL queries. The following chapter discusses the ideas behind the implementation of the presented library beginning with the mentioned program transformation translating conditions on database predicates into restrictions expressed with database specific combinators.

4. Implementation

In this chapter we discuss the implementation of the presented database library. Initially we describe a program transformation which is applied to programs utilizing database predicates. This transformation enables efficient generation of SQL queries for programs that do not use the database specific combinators by automatically introducing restrictions that can be translated into a database query. After describing this transformation, an inliner is presented, which prepares the original program for transformation. Finally, the implementation of the three key operations of dynamic predicates, assertion, query and retraction are discussed. The general idea of their implementation is given before we describe the full approach, which considers the storage of arguments in separate tables.

4.1. Transforming Dynamic Predicates

The presented database library provides combinators which can be translated into SQL queries. Conditions expressed with those combinators are therefore checked by the database system instead of being checked by the application using the library. These combinators, however, are slightly inconvenient to program with because the table's columns have to be referenced directly and thus the programmer has to consider the database used to store persistent predicates. A key advantage of persistent predicates is transparency, i.e., they can be used without knowledge of their internal storage. Using the SQL combinators directly destroys this transparency because the structure of the underlying tables has to be considered in order to use them. Furthermore, relying on a database system which is able to perform the described queries restricts their use to database predicates. Predicates stored in files are not affected by the database specific combinators; thus, changing the external storage from a relational database to, e.g., files renders the database specific combinators introduced in the program code useless.

This section describes a program transformation translating conditions on database predicates into restrictions that can be translated into an SQL query. Conditional database predicates defined using `(|>)` are extended with an equivalent restriction attached by `(.|>)`, and a warning is presented if such a translation fails. Thus, the described transformation combines the advantages of the database specific combinators with those of the more convenient generic combinators on dynamic predicates: Programs using the generic combinators are translated into programs that can be efficiently evaluated using a database system, and the storage mechanism of database predicates can later be replaced by others, since no database specific combinators are used in the original program.

The provided transformation basically includes an initialization and three different

optimizations. The initialization collects information necessary to send database queries, i.e., the database name as well as the names of the tables and their columns are stored to be employed for query generation. After the initialization the program can be used to access the underlying database. Additionally we provide three different optimizations that use the database specific combinators to add

- restrictions for conditions attached to database predicates,
- restrictions for argument values of database predicates and
- projections to query only required columns.

The first optimization generates restrictions that can be translated into SQL queries from boolean conditions attached to a database predicate. If there are ground values or shared variables as arguments of a database predicate, the second optimization uses the database specific combinators to express such a restriction. Finally, the projection combinator is used to restrict the query to required columns omitting those not needed for the definition of a database predicate.

In the remainder of this section we describe the initialization of database programs, introduce an approach to readable meta-programming and consider information employed by all optimizations and how to compute it. Then we describe each optimization in an own subsection giving typical examples for the translated expressions.

4.1.1. Initialization

While the optimizations described in Sections 4.1.5, 4.1.6 and 4.1.7 modify the generated database queries the initialization enables them, and hence, it is required to run the database program. The programmer declares database predicates giving a type signature and a storage location specifying the database and the table to store the facts. For instance, the predicate `publication` is declared as

```
data Publication = Publication Author Title
data Author = Author Name (Maybe Address)

type Title = String
type Name = String
type Address = String

publication :: Publication -> Dynamic
publication persistent "db:currydb.publications"
```

and thus, the table *publications* in the database *currydb* has to be considered. If it exists, the table's columns are queried with the SQL statement

```
DESCRIBE TABLE publications
```

and stored along with the names of the database and the table as a database table specification:

```
data DBSpec = DBSpec String String [String]
```

The specification for the table *publications* is

```
DBSpec "currydb" "publications" ["name","address","title"]
```

This information can be employed to query the facts stored in the specified table with the SQL query

```
SELECT name, address, title FROM publications
```

which is sent to the database *currydb*.

If the table *publications* does not yet exist in the database *currydb*, it has to be created with the SQL statement

```
CREATE TABLE publications (name TEXT, address TEXT, title TEXT)
```

To generate this SQL statement, the columns[1] have to be computed from the type signature of the predicate `publication`.

In Section 3.2 we described how arguments are represented in the columns of a database table. Each argument of the database predicate is stored in at least one column of the associated table; records are stored in multiple columns, one for each component. As the sole argument of the predicate `publication` is a record of type `Publication`, it is stored in more than one column. The first part of the data type `Publication` is of type `Author` which is itself a record type and, thus, stored in multiple columns. Because both arguments of the constructor `Author` are of type `String`, it is stored in two columns of type `TEXT`. As the second argument of the surrounding constructor `Publication` is also of type `String`, it is also stored in a column of type `TEXT`, and three columns are required to store values of type `Publication` as a whole.

To prepare a program which contains database predicates for database access, the associated database tables have to be examined to store the names of their columns. If there is no associated table yet for a database predicate, it can be created considering information about the predicate's type. After this initialization the program can be used; to generate more efficient queries, however, additional program transformations described in later sections are performed.

4.1.2. FlatCurry

Curry programs are compiled into a simplified core language FlatCurry that can be read by other Curry programs. In this section, we describe how FlatCurry modules and expression are represented in Curry. Curry modules are represented by the data type `Prog`. A value of this type has the form

```
Prog modname imports typedecls functions opdecls
```

where

[1] Unfortunately, the column names created automatically are not as eloquent as shown in the example.

- `modname` is the name of the module,
- `imports` is a list of names of imported modules,
- `typedecls` is a list of algebraic data type declarations,
- `functions` is a list of function definitions and
- `opdecls` is a list of infix operator declarations.

In this section, we focus on function definitions and the expressions used in function's bodies.[2] A function declaration has the form

```
Func name arity type (Rule [i_1,...,i_arity] e)
```

where

- `name` is the functions name,
- `arity` is the count of the functions arguments and
- `type` is a type signature.

This declaration represents the function definition

```
name :: type
name x_1 ... x_arity = e
```

and each `i_j` is the index of the variable `x_j`. The expression `e` in the rule's body is of type `Expr` and can be constructed in various ways. Before we consider the constructors of `Expr`, we discuss the representation of expressions more generally. The expression

```
if b then x else y
```

is represented as a call to the function `if_then_else`

```
if_then_else b x y
```

Higher order applications are represented by an external function `apply`. For instance, the body of the definition

```
f x y = x y
```

is represented as

```
apply x y
```

Conditional rules are expressed as calls to the external function `cond`. For instance,

```
zero x | x=:=0 = success
```

is represented as

```
zero x = cond (x=:=0) success
```

[2] See http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/FlatCurry.html for a complete description.

Function calls have the form

```
Comb FuncCall name args
```

where

- `FuncCall` is of type `CombType` and characterizes the call as a function call,
- `name` is the name of the called function and
- `args` is a list of expressions representing the arguments of the call.

Constructor calls are represented similarly; they employ `ConsCall`, not `FuncCall`, as comb type. Partial calls use `PartCall n` as comb type, where `n` is the count of missing arguments. Beyond function or constructor calls, FlatCurry expressions can be variables `Var n` represented by a unique index `n` or literals represented as `Lit l` where `l` is an `Integer`, `Float` or `Char` constant. There are also constructors for local declarations, possibly of free variables, for nondeterministic choices and for case distinctions.

The transformations described in the next sections modify FlatCurry programs that are represented in Curry like presented in this section. The next section describes a programming style that helps to do this in a convenient way.

4.1.3. Readable Meta-Programming

Manipulating programs with other programs is called meta-programming. FlatCurry expressions soon get hard to overlook, so this Section presents an approach to concisely describe them. Moreover, functions that allow for simulated pattern matching on FlatCurry expressions are introduced.

To get in touch with how FlatCurry is represented in Curry, consider the following expression representing the string `"Curry"`:

```
(Comb ConsCall ("prelude",":") [
  (Lit (Charc 'C')),
  (Comb ConsCall ("prelude",":") [
    (Lit (Charc 'u')),
    (Comb ConsCall ("prelude",":") [
      (Lit (Charc 'r')),
      (Comb ConsCall ("prelude",":") [
        (Lit (Charc 'r')),
        (Comb ConsCall ("prelude",":") [
          (Lit (Charc 'y')),
          (Comb ConsCall ("prelude","[]") ([]))])])])])])
```

Writing those terms gets awkward if done frequently and is typing error-prone, since strings are used to reflect function and constructor names. A both natural and sufficient solution is to introduce wrapper functions for function and constructor calls:

```
char_ c = Lit (Char c)
x :. xs = Comb ConsCall ("prelude",":") [x,xs]
nil_ = Comb ConsCall ("prelude","[]") []
```

The representation of the string `"Curry"` can now be expressed as

```
char_ 'C' :. char_ 'u' :. char_ 'r' :. char_ 'r' :. char_ 'y' :. nil_
```

To be even more concise, this expression can be generalized using the functions `map` and `foldr`:

```
string_ = foldr (:.) nil_ . map char_
```

Now the above expression changes to `string_ "Curry"` which is apparently the most concise description possible. Note that the wrapper functions take multiple arguments like the functions they represent, instead of a list of arguments like the constructors they evaluate to. Another example in Section 4.1.6 shows that this is useful allowing for partial application of the wrapper functions.

Pattern matching is a powerful mechanism to structure the definition of a function. It enables the programmer to break a function into multiple rules that can be understood independently. Matching complex FlatCurry expressions is tedious, but fortunately a similar technique can be applied to decompose FlatCurry expressions as was applied to construct them. As a both basic and elegant approach, logical search can be employed to *reverse* the presented string converter: For a given FlatCurry expression `e` the expression

```
let s free in string_ s =:= e &> s
```

evaluates to the string, represented by `e`; no additional converter function is necessary. The function `(&>)` simulates a guard and is defined as

```
(&>) :: Success -> a -> a
c &> x | c = x
```

Although logical search often suffices to elegantly decompose FlatCurry expressions, pattern matching with case expressions is more powerful in certain situations. Consider the definitions

```
x &&. y = Comb FuncCall ("prelude","&&") [x,y]
True_ = Comb ConsCall ("prelude","True") []

conjunction = foldr1 (&&.)
```

The function `conjunction` takes a non-empty list of FlatCurry expressions and constructs the boolean conjunction of the given elements. For instance the expression

```
conjunction [True_,True_,True_]
```

evaluates to

```
(Comb FuncCall ("prelude","&&") [
  (Comb ConsCall ("prelude","True") ([])),
  (Comb FuncCall ("prelude","&&") [
    (Comb ConsCall ("prelude","True") ([])),
    (Comb ConsCall ("prelude","True") ([]))])])
```

and thus equals

```
True_ &&. True_ &&. True_
```

A function `parts` that computes the parts of a boolean conjunction can be defined using logical search as

```
parts e | conjunction bs =:= e = bs
 where
  bs free
```

This function, however, nondeterministically returns all possibilities for an argument of `conjunction` that lead to the given expression, and this is probably not what the programmer needs. For instance,

```
parts (True_ &&. True_ &&. True_)
```

nondeterministically evaluates to a value equal to one of the lists

```
[True_ &&. True_ &&. True_]
[True_, True_ &&. True_]
[True_, True_, True_]
```

Only the last result corresponds to the parts of the conjunction, the others can be further divided. The given function `parts` could be alternatively defined as[3]

```
parts e | e =:= (b &&. b') = parts b ++ parts b' where b, b' free
parts e = [e]
```

To obtain the desired behavior, the second rule must not be considered if the first rule matches; information about a failing match needs to be employed to define the function appropriately. In Curry we can express this behavior using case expressions:

```
parts e = case e of
  Comb FuncCall ("prelude","&&") [b,b'] -> parts b ++ parts b'
  _ -> [e]
```

Unfortunately, at the same time we gain the opportunity to employ negative information, we lose the convenient programming style enabled by the presented combinators, because we have to write patterns on FlatCurry expressions.

To release the programmer from this burden, we define a data type for patterns on FlatCurry expressions and a function to perform the actual matching, which can be used in guarded rules of a function. Since pattern matching involves binding of unknown values we use free variables to express them. Patterns are defined as follows[4]:

[3]Unlike the previous version, this function decomposes arbitrarily nested conjunctions.

[4]We only give slightly modified parts of the implementation for documentation purpose.

```
data Pattern = V Expr | PChar Char | PComb QName [Pattern] | ...
```

The constructor `V` wraps free variables and is used to express unknown values that shall be bound by the matching function. Patterns can be defined for every kind of FlatCurry expression. As an example consider the implementation of the matching function `(@)` for the given pattern types:

```
exp @ (V x) | exp=:=x = True

exp @ (PChar c')
  = case exp of
      Lit (Charc c) -> c=:=c' &> True
      _ -> False

exp @ (PComb name' args')
  = case exp of
      Comb _ name args -> name==name' && and (zipWith (@) args args')
      _ -> False
```

The first rule binds free variables. If `(@)` is called with a ground term as first argument and in the second argument all arguments of `V` are free variables, the guard always succeeds. As `(@)` is employed for pattern matching, its first argument should always be ground. The second rule matches against characters. Note that the character argument of `PChar` should be a free variable to prevent the function from failing. These deficiencies could be overcome if a primitive function `isFree` detecting whether its argument is a free variable was available. Employing `isFree` would allow for the use of ordinary FlatCurry expressions instead of separately defined patterns and matching against certain literals would be enabled without loosing the possibility to use free variables as literals. This would allow, e.g., to match against a certain string. A function similar to `isFree` is provided in the employed Curry implementation [6]; it is, however, an unsafe feature and therefore not used in this context. The last given rule is the most interesting because patterns can be nested. To match a function or constructor call, its name is compared to the name given in the pattern and its arguments are matched recursively. To give an example how `(@)` can be used to define functions on FlatCurry expressions, we redefine the function `parts`:

```
x &&- y = PComb ("prelude","&&") [x,y]

parts e
 | e @ (b &&- b') = parts b ++ parts b'
 | otherwise = [e]
 where
  b, b' free
```

The introduced FlatCurry pattern type enables convenient pattern matching of FlatCurry expressions. At the same time, it allows for employing failing matches because the matching function has return type `Bool` rather than `Success`. Nested patterns can be written

much more concisely with the presented pattern type than with case expressions.

We presented an approach to readable meta-programming including functions to concisely construct FlatCurry expressions and a mechanism to decompose them with simulated pattern matching. As an example we discussed strings and boolean conjunctions, but the approach can be carried over to any kind of FlatCurry expression. The transformation presented in the next sections employs the given ideas, see Section 4.1.6 for another example of the given approach.

4.1.4. Associating Argument Positions with Columns

Since columns must be referenced directly using the database specific combinators, these need to be associated with argument positions of the predicate and the expressions at these argument positions. Reconsider the predicate storing publications by their authors and titles:

```
data Publication = Publication Author Title
data Author = Author Name (Maybe Address)

type Name = String
type Title = String
type Address = String

publication :: Publication -> Dynamic
publication persistent "db:pubdb.publications"
```

For an application

```
publication (Publication (Author name address) title)
```

we get the information shown in Table 4.1. If one expression, e.g. a variable, is associated

argument expression	column numbers
`name`	0
`address`	1
`title`	2

Table 4.1.: Associating Arguments with Columns

with more than one column, all column numbers would be recorded in the table. To compute this information, `publication`'s type has to be considered since it determines the columns which are required to store facts. From the type signature and the associated type declarations we can conclude that three columns are required to represent the predicate `publication` and that the first two columns are associated with the arguments of the constructor `Author` and the third column is associated with the second argument of the constructor `Publication`. Thus, analyzing the application shown above, the variable `name` corresponds to the first column, the variable `address` to the second and the variable `title` to the third column. Such tables associating argument expressions with a list of

column numbers are used in all transformations described in the next subsections. Since columns must be referenced directly using the database specific combinators, associating columns with expressions is the main problem to solve for the transformations.

4.1.5. Translating Conditions

The combinator `(|>)` can be used to attach boolean conditions to dynamic predicates. For instance, a call to the predicate `publication` could be restricted to match only those publications that contain the word "Geometry" in their title:

```
publication (Publication author title)
  |> "Geometry" `substringOf` title
```

Using the information which is provided in the table associating argument positions with columns of a database table, conditional dynamic predicates can be augmented with a restriction build from the combinators presented in Section 3.3. Variables are translated into an associated column using the function `col :: Int -> SQLExp a`, literals and strings are wrapped with `val :: a -> SQLExp a` and called functions are replaced with their database equivalent if possible. Since there is a function

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

to express the substring condition and the variable `title` is associated to column 2 (see Table 4.1), the above predicate can be translated into

```
publication (Publication author title)
  .|> substringOf' (val "Geometry") (col 2)
```

If the programmer defines his own function `mySubstringOf` to express the substring condition, this cannot be translated into an efficient version; so the expression

```
publication (Publication author title)
  |> author == (Author "Euclid" (Just "Alexandria"))
  && "Geometry" `mySubstringOf` title
```

is translated into

```
publication (Publication author title)
 .|> col 0 .== val "Euclid" .&& col 1 .== val "Alexandria"
  |> "Geometry" `mySubstringOf` title
```

and a warning is presented, indicating that

```
"Geometry" `mySubstringOf` title
```

could not be translated.

Conditional database predicates can be automatically translated into a form that allows for efficient database query generation using information about which expressions are associated with which columns of the underlying database tables. Variables are translated into a reference to an associated column, literals and strings are wrapped as values of the internal `SQLExp` data type and function calls are translated into equivalent translatable calls if possible.

4.1.6. Restricting Columns with Values of Arguments

Conditions, however, are not the only source of restrictions generated for calls to database predicates. For instance, the programmer can use shared variables to express equalities among database columns. Section 3.1.4 introduced a query for titles of publications written in Alexandria:

```
writtenInAlexandria :: Title -> Dynamic
writtenInAlexandria title
  = publication (Publication name title) <>
    author (Author name (Just "Alexandria"))
 where
  name free
```

The information about the shared variable `name` can be translated into SQL as well as the string `"Alexandria"` which is associated to the second column of the table *authors*. The information about the column associations for the combined predicate is shown in Table 4.2. The variable `name` is shared among the calls of the database predicate

argument expression	column numbers
`name`	0,2
`title`	1
`"Alexandria"`	3

Table 4.2.: Associating Arguments with Columns

and, thus, has two associated columns. The variable `title` is associated with column 1 and the string `"Alexandria"` with column 3. Thus, an efficient query for the combined predicate should restrict column 3 to equal `"Alexandria"` and columns 0 and 2 to contain the same value, since otherwise if n publications and m authors are stored, the query will have $n * m$ answers. Hence, the combined predicate is translated into

```
publication (Publication name title) <>
author (Author name (Just "Alexandria")) .|>
col 0 .== col 2 .&& col 3 .== val "Alexandria"
```

To give an example how the ideas presented in Section 4.1.3 can be employed to generate the translatable conditions, consider the following example code[5]:

```
generateRestriction (arg,cols)
  | isVar arg && 1 < length cols
    = let (col:cols') = cols
       in foldr1 (.&&.) (map (col .==.) cols')

  | isGround arg = foldr1 (.&&.) (map ((val_ arg) .==.) cols)
```

[5]The code is modified to point out the general approach instead of quoting the actual implementation.

The predicates `isVar` and `isGround` can be applied to FlatCurry expressions. They hold for those expressions representing variables or ground terms respectively. The functions `(.&&.)`, `(.==.)` and `val_` construct calls to the corresponding functions without attached period or underscore. The type signature of `generateRestriction` is omitted because all involved expressions represent FlatCurry expressions. Hence, they are of the same type `Expr` and the type signature is of little use. The function's argument corresponds to one row of the shown tables associating arguments with a list of column numbers. If the argument is a variable associated with more than one column, equality restrictions for those columns are generated. If the argument is a ground value, all associated columns are restricted to equal that value. Without using the wrapper functions we could hardly use higher order functions to build the restrictions, because partial applications of FlatCurry expressions representing function calls could only be expressed by lambda abstractions. Hence, beyond documenting the approach of generating restrictions according to arguments, this example shows how the ideas presented in Section 4.1.3 help to write readable code in meta-programming.

4.1.7. Adding Projections

The presented restrictions control the `WHERE`-part of the generated SQL query. Another combinator `(.!!)` is provided to declare projections which hence controls its `SELECT`-part. To automatically compute those columns that need to be queried, again, the information about the associated columns is used. To demonstrate how to compute which columns need to be projected, we consider an academic example: In the definition of `q`

```
p :: Int -> Bool -> Int -> Dynamic
p persistent "db:currydb.p"

q :: Int -> Dynamic
q a = p a (even b) b where b free
```

all columns of the table storing `p`'s facts have to be queried but all for a different reason: The first column is required because the variable `a` is a pattern variable of `q`. The third column has to be queried because the variable `b` is also the argument of the call to the function `even`. The result of this call has to be compared to the value in the second column, so this is also required. There are three different reasons for a column to be projected:

- It is associated to a pattern variable of the predicate being defined,
- it is associated to a complex expression, e.g., a function call or
- it is associated to a variable which is part of a complex expression, e.g., an argument of a function call.

If a column is associated to a pattern variable, this variable is visible outside the defined predicate and has to be bound by the query. If it is associated with a complex expression,

it has to be checked against the queries result; and finally, if it is part of a complex expression, it has to be bound to evaluate this expression. A column is not required if it is associated to a variable which is not used somewhere else in the program.

In this section we discussed the program transformation which applies three different optimizations of calls to database predicates. All optimizations rely on information about expressions associated to columns of database tables which can be computed considering the type signature of database predicates. The presented restrictions often significantly limit the answers returned from the generated database query, and thus, this transformation is crucial for reasonable run-time performance.

4.2. Inlining Curry Programs

The program transformation described in Section 4.1 expects combined dynamic predicates to be inlined. Inlining replaces function calls with function's right-hand-sides, hence brings forward reduction steps that would have been performed at runtime otherwise. For instance, consider the definition of the function `f`

```
f x = 2*x
```

An inlining step for `f` consists of replacing a call of `f` by an instance of `f`'s body. So the expression

```
f (a+b)
```

is translated into

```
2*(a+b)
```

by inling the call of the function `f`.

The inliner, used to prepare a program which includes database access, is inspired by the Glasgow Haskell Compiler inliner [9]. Although Curry's core language FlatCurry differs from HaskellCore, some key ideas of the Haskell inliner, especially the notion of loop-breaking functions described below, can be applied to Curry as well.

The employed algorithm can be described as follows:

```
inlining process :
  while program changes do
    perform inlining steps
    perform simplifications
    done
```

Inlining steps are performed as long as they change the program, and they are interleaved with minor simplifications like dead code elimination and others described below. An inlining step is only performed if the function's patterns can be matched. For instance, a call to the function

```
null [] = True
null (_:_) = False
```

is only inlined if its argument is a constructor rooted term. The simplifications comprise

- dead code elimination,
- elimination of calls to the primitive function `apply` and
- case simplification.

Dead code elimination deletes definitions of functions that are neither used nor exported. A function definition becomes dead code if every call to the function has been inlined. In FlatCurry higher order applications are expressed with a primitive function `apply` (cf. Section 4.1.2). However, if the first argument of apply is an evaluated partial function call, then we can augment the function's arguments with the second argument of apply instead of using apply. This situation is quite common after inlining, and new inlining steps could be enabled by this transformation, since only function applications with saturated arguments can be inlined.

There are different modifications on case-expressions which play together to simplify an inlined program. If the scrutinee of a case-expression is a variable and used somwhere in the branch expressions, we can replace it with the matched expression and possibly enable the transformation described next for another case-expression. For instance,

```
case x of
  Nothing -> []
  Just y -> case x of
              Nothing -> y
              Just z -> z
```

can be transformed into

```
case x of
  Nothing -> []
  Just y -> case Just y of
              Nothing -> y
              Just z -> z
```

If the scrutinee is known to be a specific literal or constructor-application, we can replace the case expression by the matching branch expression. We can simplify the above example:

```
case x of
  Nothing -> []
  Just y -> y
```

If the scrutinee of a case expression is itself a case expression, and all branch expressions of the inner case are literals or constructor-applications, we can merge the branches of both case expressions. For instance,

```
case (case x of
       Nothing -> []
       Just y -> [y]) of
  [] -> False
  (z:zs) -> True
```

can be transformed into

```
case x of
  Nothing -> False
  Just y -> True
```

The presented transformations play well together to simplify an inlined program. Often, they give rise to further inlining steps.

The example program presented below uses higher order functions to combine dynamic predicates. The predicate `wroteTheSame` holds for two different authors that wrote publications with the same title.

```
data Publication = Publication Name Title

publication :: Publication -> Dynamic
publication persistent "db:currydb.publications"

publications :: [Publication] -> Dynamic
publications = foldr1 (<>) . map publication

wroteTheSame :: Name -> Name -> Dynamic
wroteTheSame name name' = publications
  [Publication name title, Publication name' title] |>
  name /= name'
 where
  title free
```

The most notable point in this example is the predicate `publications` being defined using recursive higher order functions and, hence, not being directly associated to a database table.

To generate SQL queries from combined dynamic predicates, these need to be inlined. Otherwise it would not be possible to determine the correct tables or column numbers that represent the predicate's data. Only those predicates that are declared persistent in the program code have associated information about the database table and its columns that store the defining facts. Those predicates, however, that are built from others using the provided combinators do not have such associated information but store their data in the tables of the underlying primitive predicates. Hence, inlining uncovers which primitive predicates were used to combine an arbitrary predicate by replacing the combinators used to construct the predicate.

Inlining the call of the predicate `publications` results in the following definition of the predicate `wroteTheSame`:

```
wroteTheSame name name'
  = publication (Publication name title) <>
    publication (Publication name' title) |>
    name /= name'
 where
  title free
```

The predicate `publication` is declared persistent in the program code; thus, its arguments can be associated to columns as described in Section 4.1.4.

In some cases a function call must not be inlined to ensure the sharing of expensive or nondeterministic computations and to prevent the inlining process from running into an infinite loop, since inlining steps are performed as long as there are matching calls of function rules that can be replaced by an instance of the rules body. Replacing all matching function calls once is called an *inlining pass* and in the *inlining process* inlining passes are repeated until they do not change the program, i.e., until no more function calls can be inlined.

4.2.1. Sharing and Nondeterminism

An argument variable of a function is called *shared* when it occurs more than once in the function's body. As an example for a function with a shared argument variable consider the definition of the function `double`:

```
double :: Int -> Int
double n = n + n
```

Assuming `f x` as an expensive computation, the call `double (f x)` must not be inlined or the resulting code may perform badly. Another problem results from nondeterministic computations. As `0?1` nondeterministically returns either `0` or `1` the call `double (0?1)` results in either `0` or `2` because `0?1` is shared as argument of `double`. The expression `(0?1) + (0?1)`, however, nondeterministically returns `0,1,1` or `2`, and thus, inlining the call `double (0?1)` would change the results the expression evaluates to.

As a consequence, calls of functions with shared arguments may only be inlined if the shared arguments may be safely duplicated. An argument may be safely duplicated if it is a variable, a literal or a partial function call whose already supplied arguments may be safely duplicated. For instance, the call `map (0?) [1,2]` may be safely inlined although the first argument of `map` is shared and `(?)` is nondeterministic. Ground terms may also be safely duplicated; we avoid duplicating them, however, to prevent immoderate code duplication.

4.2.2. Termination

Inlining recursive functions does not terminate in general. To prevent the inlining process from running into infinite loops, the call-graph of the functions that may be inlined has to be considered. A *call-graph* for a set of functions is a graph which has a directed edge from a function `f` to a function `g` iff `g` is applied somewhere in the body of `f`.

So, if there is a circle in the call-graph, inlining all functions on the circle will not terminate. To assure termination, one can choose a *loop-breaking* set of functions such that the modified call-graph where this set of functions is removed has no circles. Not inlining any loop-breaking function effectively ensures termination. Note that every loop-breaking set of functions contains all directly recursive functions, and thus, refusing to inline loop-breaking functions means refusing to inline any directly recursive function.

There are, however, examples where inlining directly recursive functions would be beneficial. We showed in the example that `map` and `foldr1` can be used to combine a non-empty list of values as arguments of a dynamic predicate. If `p` is a dynamic predicate taking exactly one argument, then

```
foldr1 (<>) . map p
```

is a function taking a nonempty list of values of `p`'s argument type and returning a conjunction of applications of `p` to every element of the given list. Refusing to inline `map` and `foldr1` would inhibit translating any so combined predicate into an efficient SQL query.

The example shows that inlining directly recursive functions often terminates and that refusing to inline them is too restrictive for the purpose of the presented database library. To describe such function calls that can be safely inlined repeatedly, we introduce the notion of descending argument positions: An argument position n of a function `f` is called *descending* iff for every call of `f` in `f`'s body there is a strict sub-term of `f`'s nth argument at the nth argument of the call. To clarify this notion look at the definition of `map`:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

`map`'s second argument is descending, since there is a strict sub-term of this argument at the second argument of the sole recursive call. For an example of a terminating function without descending arguments consider `a`:

```
a [] y = ():y
a (_:xs) [] = a xs [()]
a (x:xs) (_:ys) = a xs (a (x:xs) ys)
```

Neither the first nor the second argument is descending because the underlined arguments are no strict sub-terms of the original arguments. Note that those calls of directly recursive functions where there is a constructor-rooted term at a descending argument position can be inlined without running into infinite loops. The presented inliner refuses to inline calls to loop breaking functions except for calls to directly recursive functions with constructor-rooted terms at a descending argument position. This ensures termination of the inlining process described above.

4.3. Asserting Facts

To assert a dynamic predicate which is declared persistent in the program's source, a string representation of the predicate's arguments has to be inserted into the associated columns of the database table used to store the predicate's facts. To assert a fact for a persistent predicate that does not use separate tables, we proceed as follows:

```
assert :
  table = get associated table
  args = convert argument values
  sql_insert( table, args )
```

Initially, the arguments of the involved predicates are converted. Primitive values are converted into strings, and records are split into multiple parts that are converted independently. The values are inserted into the table associated to the persistent predicate.

Consider the predicate `publication` storing one author and the title of a publication:

```
data Publication = Publication Author Title
data Author = Author Name (Maybe Address)

type Title = String
type Name = String
type Address = String

publication :: Publication -> Dynamic
publication persistent "db:currydb.publications"
```

The facts for this predicate are stored in a table with three columns created by

```
CREATE TABLE publications (name TEXT, address TEXT, title TEXT)
```

To assert the book "Elements" by Euclid of Alexandria, the IO action

```
assert (publication
  (Publication (Author "Euclid" (Just "Alexandria")) "Elements"))
```

has to be performed. The corresponding SQL statement is

```
INSERT INTO publications VALUES ("Euclid","Alexandria","Elements")
```

and the resulting table is shown in Table 4.3.

name	**address**	**title**
Euclid	Alexandria	Elements

Table 4.3.: Table *publications* in database *currydb*

Alternatively, the author of a publication can be stored in a separate table along with a unique reference. In this case two tables are created to store `publication`'s facts by

```
CREATE TABLE publications (author INT, title TEXT)
CREATE TABLE authors (ref INT, name TEXT, address TEXT)
```

To perform the assertion shown above, a new reference has to be computed, and all values have to be inserted in the corresponding tables. In general, a persistent predicate with associated separate tables is asserted as follows:

```
assert :
  table = get associated table
  sepTables = get associated separate tables
  args = convert argument values

  for each t in sepTables do
    ref = get_unused_reference( t )
    sepArgs = get those args stored in t
    sql_insert_with_ref( t, ref, sepArgs )
    args = replace sepArgs with ref in args
  done

  sql_insert( table, args )
```

Separate tables are considered additionally to the predicate's main table. For each separate table, the corresponding arguments are stored with a new reference. This reference is stored to be inserted in the original table associated to the persistent predicate.

We mix SQL with Curry code to describe how the values in the example are inserted:

```
SELECT MAX(ref) FROM authors

let newref = maxref + 1

INSERT INTO authors VALUES (newref, "Euclid", "Alexandria")
INSERT INTO publications VALUES (newref, "Elements")
```

Initially, a new reference is computed by querying the previously inserted reference. Finally, this reference is used to insert the appropriate values into the tables *authors* and *publications*. Note that the whole assertion has to be atomic to prevent multiple use of references in parallel assertions. Transactions discussed in Section 3.1.3 can be employed to ensure mutual exclusion of parallel assert operations. Tables 4.4 and 4.5 show the results of the SQL statements.

author	**title**
1	Elements

Table 4.4.: Table *publications* in database *currydb*

ref	**name**	**address**
1	Euclid	Alexandria

Table 4.5.: Table *authors* in database *currydb*

4.3.1. Lists as Arguments

To describe how lists are asserted, we reconsider the example employed in Section 3.2.3:

```
data Publication = Publication Id [Author] Title [Id]
data Author = Author Name (Maybe Address)

type Id = Int
type Title = String
type Name = String
type Address = String

publication :: Publication -> Dynamic
publication persistent "db:currydb.publications"
```

Facts of the predicate `publication` are stored in three tables, since the lists of authors and references are stored in extra list tables (cf. Section 3.2.3). The employed tables are created by

```
CREATE TABLE publications (id INT, authors INT, title TEXT, refs INT)
CREATE TABLE authors (ref INT, idx INT, name TEXT, address TEXT)
CREATE TABLE references (ref INT, idx INT, id INT)
```

List tables have to be treated differently compared to other tables storing data corresponding to one fact in each row, since in a list table multiple rows correspond to the same fact. These rows are inserted at once, and a reference is computed similar to the assertion of a single argument in a separate table. A *null* value is inserted instead of a reference if the argument list is empty.

In general, list tables are handled similar to separate tables:

```
assert :
  table = get associated table
  listTables = get associated list tables
  args = convert argument values

  for each t in listTables do
    ref = get_unused_reference( t )
    listArgs = get list stored in t from args
    sql_insert_all_with_index( t, ref, listArgs )
    args = replace listArgs with ref in args
  done

  sql_insert( table, args )
```

For each list a new reference is computed which is stored in the corresponding table together with the list elements.

The following IO action asserts Euclid's Elements with an additional identifier and a list of references:

```
assert (publication (Publication 17
  [Author "Euclid" (Just "Alexandria")] "Elements" [42,7,11]))
```

Initially, the lists are asserted with new references, and finally, these references are inserted into the corresponding columns of the table *publications*.

id	authors	title	refs
17	1	Elements	1

Table 4.6.: Table *publications* in database *currydb*

```
SELECT MAX(ref) FROM authors
let newref0 = maxref + 1
INSERT INTO authors VALUES (newref0,0,"Euclid","Alexandria")

SELECT MAX(ref) FROM references
let newref1 = maxref + 1
INSERT INTO references VALUES (newref1,0,42),(newref1,1,7),(newref1,2,11)

INSERT INTO publications
VALUES (17,newref0,"Elements",newref1)
```

ref	idx	name	address
1	0	Euclid	Alexandria

Table 4.7.: Table *authors* in database *currydb*

Tables 4.6, 4.7 and 4.8 show the values inserted in the database tables.

ref	idx	id
1	0	42
1	1	7
1	2	11

Table 4.8.: Table *references* in database *currydb*

If more than one fact has to be inserted as a combined dynamic predicate, SQL statements can be accumulated if they store data in the same tables. Similar to the elements of an argument list, multiple facts defining the same predicate can be inserted at once if they do not use extra tables to store parts of their associated data. This can significantly reduce the count of SQL statements that have to be sent to the database.

This section revealed how different types of arguments of persistent predicates are asserted by inserting string representations of the predicate's arguments into the columns of the associated tables. Additional tables separately storing some arguments and list tables storing the elements of an argument list in multiple columns of an extra table were considered.

4.4. Querying Dynamic Predicates

To retrieve stored information about a dynamic predicate, the database is queried, and the answers are converted corresponding to the predicate's argument types. The query is generated from the associated information about the tables and their columns and from a possibly attached condition restricting the query.

Simple persistent predicates, i.e., those that do not use additional tables to store their arguments, can be translated into a single SQL query. To query the arguments of a simple database predicate we proceed as follows:

```
query :
  table = get associated table
  proj = get projected columns
  restr = get attached restriction
  args = sql_select( table, proj, restr )
  convert args according to argument types of predicate
```

First, we query the predicate's values from the associated database table. Database specific conditions and projections are employed to limit the requested values. Then the requested values are converted according to the argument types of the persistent predicate.

Reconsider our running example that does not involve lists as arguments:

```
data Publication = Publication Author Title
data Author = Author Name (Maybe Address)

publication :: Publication -> Dynamic
publication persistent "db:currydb.publications"
```

The associated database table is omitted since it was given several times in previous sections (cf. Table 3.2 or 4.3). The predicate

```
workedOnGeometry :: Name -> Dynamic
workedOnGeometry name
  = publication (Publication (Author name X) X)
    .|> substringOf' (val "Geometry") (col 2)
    .!! [0]
```

involves both a condition that can be translated into a database query and a projection of the first column of the table, which represents the name of the publication's author.

The corresponding SQL query is

```
SELECT name FROM publications WHERE title REGEXP "Geometry"
```

If the predicate is a combination of other persistent predicates, these are combined into one query.

It is still possible to translate even a combined dynamic predicate into a single query if some arguments are stored in separate tables. To express the references used to associate the relative values, additional conditions have to be generated. Recall the

Tables 4.4 and 4.5 and the above query for authors that worked on geometry. The latter has to be changed by adding a condition which associates the column *author* of the table *publications* with the column *ref* of the table *authors*:

```
SELECT authors.name
  FROM publications, authors
 WHERE publications.author=authors.ref
   AND publications.title REGEXP "Geometry"
```

4.4.1. Lists as Arguments

Due to the different nature of list tables compared to those that store other arguments, one query is not enough if the predicate has a list as argument stored in such a table. In list tables an argument for a single fact is usually stored in multiple rows, which is not the case for other tables. Hence, an extra query for every queried list table is necessary. Initially, a query is generated like described before, not yet heeding that some arguments may be lists. Then, those values that are references into list tables are used to query these tables, and the results are converted into a list matching the corresponding argument type of the predicate. We return to our running example which involves lists for authors and references:

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

publication :: Publication -> Dynamic
publication persistent "db:currydb.publications"
```

The expression `getDynamicSolutions publication` is translated into the SQL queries

```
SELECT id, authors, title, refs FROM publications
SELECT name, address FROM authors WHERE ref=1 ORDER BY idx
SELECT id FROM references WHERE ref=1 ORDER BY idx
```

The condition in the second and third query is `ref=1` because the values of both columns *authors* and *refs* in the first query is `1` (see Table 4.6). If there is more than one matching entry in the column referencing a list table, a query for every entry is generated since a different list is referenced by every entry. To preserve their order, the stored lists are queried sorted by their index.

Unfortunately, this approach to query lists can be slightly inefficient if many lists have to be queried since for every list an extra query is generated. Thus, it would be beneficial to restrict queried lists in advance. Since they are stored in multiple rows of an extra table, this is not as simple as restricting other values. In Curry, the function

```
elem :: a -> [a] -> Bool
```

is used to check whether a list contains some value. Consider a query for publications that reference Euclid's Elements:

```
referencesElements title
  = publication (Publication X X title refs)
  |> 17 `elem` refs
 where
  refs free
```

Imagine a large database with a few publications referencing Euclid's Elements indicated in the Tables 4.9, 4.10 and 4.11. The query presented above would retrieve *every* publi-

id	authors	title	refs
...	...	...	...
17	42	Elements	71
...	...	...	...
18	99	Foundations of Geometry	1899
...	...	...	...

Table 4.9.: Table *publications* in Database *currydb*

ref	idx	name	address
...	...	...	...
42	0	Euclid	Alexandria
...	...	...	...
99	0	Hilbert	NULL
...	...	...	...

Table 4.10.: Table *authors* in Database *currydb*

ref	idx	id
...	...	...
71	0	42
71	1	7
71	2	11
...	...	...
1899	0	17
...	...	...

Table 4.11.: Table *references* in Database *currydb*

cation in the database and test if its references contain the identifier `17` *afterwards*. To be able to restrict lists *in advance*, the function

```
elem' :: SQLExp a -> SQLExp [a] -> SQLExp Bool
```

is provided which takes a literal or string lifted to the `SQLExp` type as first argument and a column reference, pointing to a list column as second. Like the other combinators it

is automatically introduced by the program transformation if possible. The restriction on lists is implemented with an extra query which is sent before the others to restrict possible references in the list table:

```
SELECT ref FROM references WHERE id=17
SELECT title FROM publications WHERE refs=1899
```

Due to the restriction, no list of references has to be queried instead of every single list of references in the database. If the references were projected in the second query, only those lists that contain a reference to Euclid's Elements would have to be queried.

In general, lists are queried as follows:

```
query :
  table = get associated table
  listTables = get associated list tables

  for each t in listTables do
    if t is restricted then do
      restr = getRestriction( t )
      refs[ t ] = sql_select_refs( t, restr )
    done
  done

  args = sql_select_with_refs( table, refs )

  for each listRef in args do
    t = get table that listRef points to
    listElems[ t ] = sql_select_values( t, listRef )
  done

  args = replace listRefs with corresponding list elements in args

  convert args according to argument types of predicate
```

A query for a persistent predicate with lists as arguments can be divided into three phases:

- Query references for lists that satisfy a given `elem`-condition.
- Use this references to query the other arguments of the predicate.
- Query elements of lists according to the former query.

Initially, list references are queried according to `elem`-conditions, given by the programmer. These references are used to restrict the rows that are queried from the main table of the predicate. Finally, the elements of those lists, that satisfy the whole query are retrieved. If the former query would not be restricted with the list references pointing to

lists that satisfy a given `elem`-condition, the last phase would retrieve much more lists, that would have to be tested afterwards.

This section presented a mechanism to query a database predicate. Beyond simple database predicates those using extra tables for their arguments where considered as well as a special kind of table storing lists.

4.5. Retracting Facts

Compared to inserting and querying data for a persistent predicate, deleting it is a bit more involved. For every table associated to a predicate or argument a DELETE-statement is generated. The problem is, however, which rows to select for deletion. This problem is discussed after the general approach for deletion is presented reconsidering our running example that does not involve lists:

```
data Publication = Publication Author Title
```

To retract Euclid's Elements, the IO action

```
retract (publication (Publication
 (Author "Euclid" (Just "Alexandria")) "Elements"))
```

has to be performed. If there are no separate tables for arguments, retracting a fact is done by constructing an appropriate condition for the DELETE-statement, since rows of a table cannot be deleted directly but only via a condition restricting the rows to delete. Thus, to retract Euclid's publication, the following query can be sent to the database:

```
DELETE FROM publications
WHERE name="Euclid" AND address="Alexandria" AND title="Elements"
```

The general approach to retraction of simple predicates is:

```
retract :
  table = get associated table
  args = convert argument values
  restr = restrict columns to equal args
  sql_delete( table, restr )
```

Those rows that store values representing the arguments of the persistent predicate are deleted from the associated database table. A condition is generated accordingly from the converted argument values.

4.5.1. Lists as Arguments

If the persistent predicate has lists as arguments, these have to be deleted separately. To determine which lists have to be deleted, a SELECT-statement is necessary. Instead of publications involving lists we consider an academic example to reveal this necessity:

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

main = do
  assert (p 1 [1,2,3] <> p 2 [1,2,3] <> p 1 [1,2,4])
  retract (p 1 [1,2,3])
```

The tables associated with `p` are shown after asserting the facts, and thus, they contain the entries for `p 1 [1,2,3]`. The rows which have to be deleted to retract `p 1 [1,2,3]`

col0	col1
1	1
2	2
1	3

Table 4.12.: Table Associated with `p`

ref	idx	col0
1	0	1
1	1	2
1	2	3
2	0	1
2	1	2
2	2	3
3	0	1
3	1	2
3	2	4

Table 4.13.: Table Associated with Second Argument of `p`

have been accentuated. The rows of one table, however, cannot be deleted without querying the other: If from the first table all lines that contain `1` in the first column were deleted, `p 1 [1,2,4]` would be wrongly retracted and the rows with reference `3` in the list table would be no longer referenced. The list table needs to be queried to test the list referenced by the second column. On the other hand, if from the list table all rows would be deleted that represent the list `[1,2,3]`, the list argument of `p 2 [1,2,3]` would be deleted resulting in an inconsistent first table, since reference `2` would no longer be present in the list table. The list cannot be removed from the list table without considering the corresponding first argument of `p` which can only be revealed by querying the first table. Hence, first the rows have to be queried similar to Section 4.4 and then the list argument is tested to find out the rows to delete:

```
SELECT col1 FROM p WHERE col0=1
```

This query has two results: `1` and `3`, and for both references the list table has to be queried:

```
SELECT col0 FROM plist WHERE ref=1 ORDER BY idx
SELECT col0 FROM plist WHERE ref=3 ORDER BY idx
```

After testing the results, the correct rows can be deleted:

```
DELETE FROM p WHERE col0=1 AND col1=1
DELETE FROM plist WHERE ref=1
```

Thus, the general approach to delete lists as arguments of persistent predicates can be described as follows:

```
delete :
  table = get associated table
  listTables = get associated list tables

  args = convert non-list argument values
  restr = restrict columns to equal args

  perform query algorithm with restr

  refs = get queried references of given list argument values
  refRestr = restrict list columns to equal a corresponding ref

  sql_delete( table, restr && listRestr )

  for each ref in refs and corresponding t in listTables do
    sql_delete_with_ref( t, ref )
  done
```

Initially the arguments of the persistent predicate are queried from the database table. An additional restriction is generated from the non-list arguments of the predicate. Then, the queried lists are tested to equal the lists given as arguments of the predicate. The references from those lists, that match the supplied arguments are employed to delete the appropriate rows from the main table associated to the predicate. Finally, the references are used to delete the lists from the associated list tables.

Retracting a fact involves both SELECT- and DELETE-statements if the predicate is stored in multiple tables. A condition has to be computed restricting the values of all columns, since no direct deletion of values is possible with SQL.

4.6. Sending SQL Queries

The SQL queries generated by the presented library need to be sent to the database with a low level primitive that takes an SQL string as argument and returns a list containing the results of the query. This primitive has the type

```
queryDB :: String -> String -> [[String]]
```

and hence, it takes the database name and the SQL query as arguments and returns the results as list of rows, which are themselves lists of cells, one for every column. It is implemented with a call to the command-line application of MySQL[6] which is called in batch mode and accessed via stdin and stdout.

In this chapter we described the ideas behind the implementation of the presented database library. The program transformation that enables the programmer to access a database both transparently and efficiently was considered. Transparency is achieved by the use of persistent predicates without database specific combinators. Efficiency is obtained by augmenting the original program with those combinators to enable the translation into most restrictive possible database queries. After considering the transformation, the three key operations of persistent predicates, i.e., assertion, query and deletion, were discussed; first mentioning the simplified approach restricted to simple predicates without more than one associated table and, finally, describing the general approach including tables for separate arguments and lists.

[6]http://www.mysql.com/

5. Evaluation

5.1. Simulating Dynamic Knowledge

The interface of the dynamic predicate library presented in [7] 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 holding for a route between two cities:

```
connection :: (City,City) -> Dynamic
connection dynamic

route :: (Dynamic -> Success) -> [City] -> Success
route known [c1,c2] = known (connection (c1,c2))
route known (c1:c2:cs)
  = known (connection (c1,c2)) & route known (c2:cs)

main = do
  known <- getKnowledge
  getAllSolutions (route known) >>= mapIO_ print
```

If `connection` 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 representing the facts of the database predicate, `route` can be defined similar to the example above:

```
connection :: (City,City) -> Dynamic
connection persistent "db:currydb.connections"

data Dyn = Connection (City,City)

route :: (Dyn -> Success) -> [City] -> Success
route known [c1,c2] = known (Connection (c1,c2))
route known (c1:c2:cs)
  = known (Connection (c1,c2)) & route known (c2:cs)

main = do
  connections <- getDynamicSolutions connection
  let known (Connection c) = c =:= foldr1 (?) connections
  getAllSolutions (route known) >>= mapIO_ print
```

The explicit definition of the function `known` is still possible in more complex examples (see Appendix B for the idea and the complete example), 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.

5.2. 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 tests.

As basis for our tests we employ a database of publications obtained from the CiteSeer[1] database. CiteSeer is a library and search engine primarily for literature in computer science. Its database can be downloaded in XML format and we have converted and stored it 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 runtime. 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

[1] http://citeseer.ist.psu.edu/

fit in memory. So, this imbalance is justified with regard to practical aspects. In Section 5.2.2 we discuss how the different amounts of stored data and different counts of returned results affect the response time of both implementations.

We now describe the employed queries and compare their run time in both tested 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 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.

5.2.1. The 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. See Appendix C for the complete module that interfaces the CiteSeer database.

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

```
intMatch title
  = publication (Publication 10000 title X)
```

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 X title X)
  |> "Curry" `substringOf` title
```

All publications that contain the word "Curry" in their title are requested.

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

```
stringMatch' title
  = publication (Publication X title X)
  |> "Database" `substringOf` title
```

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 X title refs)
  |> 3264 `elem` refs
 where
  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.

5.2.2. The Results

Table 5.1 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

Condition	**Files : (20,000)**		**Database : (100,000)**	
	count :	seconds	count :	seconds
`id is 10000`	1 :	0.7	1 :	0.4
`"Curry" in title`	0 :	11.7	5 :	1.3
`"Database" in title`	367 :	11.2	1758 :	15.3
`3264 in refs`	1 :	1.3	4 :	0.8

Table 5.1.: Results of the Performance Comparison

implementations in less than a 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 database implementation could benefit from an index on the database tables if these grow.

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 back-end for the database implementation potentially increases its performance substantially.

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.

6. Related and Future Work

The theoretic foundations common to logic programming and deductive databases are covered in [4]. It is shown that the fields of deductive databases and logic programming can be studied uniformly from a single point of view.

The notion of persistent predicates is introduced in [3] where a database implementation based on a Prolog to SQL compiler [5] is provided. Persistent predicates are implemented by storing facts in both files and relational databases. They 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. The database implementation presented in [3] relies on a mid-level database layer which handles database access sophisticatedly compared to the implementation presented in this thesis. While a mid-level database layer enables standardized access to different database API's, our implementation communicates through standard IO with a command line application for database access. In [3] clustering of Prolog queries is performed to access the database more efficiently. The approach presented in this thesis employs special conjunction operators on dynamic predicates for the same purpose. We also provide a mechanism to handle transactions which is especially important for developing web applications. In such applications one does not know when the individual programs reacting to clients requests are executed. For database access, [3] relies on side effects, i.e., non-declarative features of Prolog. Due to side effects, a computed result depends on the employed evaluation strategy.

Evaluation strategies of functional logic programming languages [1] cause problems for the evaluation of persistent predicates. Their evaluation order is difficult to determine in advance by the programmer and the order of database updates and queries is important for the programs behavior. These problems are solved in [7] by the use of monads [12], which provide a clear separation of the declarative and the imperative part of a program. The use of monads is an advantage over the Prolog implementation, since declarative and imperative parts of database programs are separated in a clean manner. We extend the library presented in [7] with a database implementation of persistent predicates. The original library provides a function to conceptually retrieve all currently known facts of arbitrary predicates. This function is not supported by the presented implementation, but can be simulated as shown in Section 5.1.

A combinator library for Haskell, which is used to construct database queries with relational algebra, is provided by [10]. It allows for a syntactically correct and type-safe implementation of database access. A general approach to embed domain specific languages into higher-order typed languages is provided and applied to relational algebra to access relational databases. The employed back-end relies on ActiveX Data Objects and, hence, can access any ODBC compliant database which is an advantage over the

presented database predicate library. The key advantage of the approach presented in this thesis, however, is a programming style which functional logic programmers are used to. While [10] *syntactically* integrates a domain specific language into the Haskell programming language, our approach *transparently* integrates database access into an existing programming paradigm. Functional features, e.g., higher order functions, can be used to construct complex persistent predicates which extend the notion of predicates common to logic programmers.

As future work the low level interface of the presented library needs to be considered: The current implementation uses a command line interface to MySQL to send database queries and retrieve results via stdin and stdout. This implementation suffices for a proof of concept but suffers from moderate performance. It should be exchanged by an interface implemented, e.g., using JDBC to be able to access arbitrary ODBC-compliant database systems. Such an implementation could provide query results lazily similar to the function `readFile` that lazily returns the contents of a text file. In the current implementation, all answers to a database query are provided at once which causes delays for large result sets. Querying results as they are needed by the application could reduce the response time of the low level database interface.

In Section 3.2.2 we presented an approach to store variant records and recursive data types in multiple tables of a database. An implementation of this approach could reveal, whether it is beneficial.

The ideas presented in Section 5.1 can be employed to automatically generate queries to compute dynamic knowledge returned by the function `getKnowledge`. Thus, a future implementation could offer the complete interface to dynamic predicates presented in [7].

7. Conclusion

Application programs need to store data that persists multiple runs of the program. Such data can be stored in files or, if large amounts of data have to be managed, in databases. Persistent predicates are an abstraction providing transparent access to persistently stored data, i.e., application programs can use persistent predicates regardless of their storage mechanism which can be flexibly exchanged.

In this thesis we provide an implementation of persistent predicates based on relational databases for the functional logic programming language Curry. It is based on the dynamic predicate library introduced by [7] and implements an alternative storage mechanism for persistent predicates. Facts are stored in a relational database, not in files, and accessed via SQL statements generated from database specific combinators.

To interface with existing databases, declarations of persistent predicates can be automatically generated. A type signature for a persistent predicate is obtained from a database table description. Arguments of a persistent predicate are stored in the database in a flat form, and, by default, each column of the database table is associated to one argument of the generated persistent predicate. The programmer can change the generated type signature to provide additional structure to the arguments. For instance, he can introduce records to store multiple columns of the table at once. We describe exactly how different kinds of arguments can be represented in columns of a database table, and we provide beneficial storage mechanisms especially for record types and lists.

The presented library aims at providing a mechanism allowing for both transparent and efficient access to relational databases by transforming storage independent programs into database specific ones. Transparent access is achieved, since persistent predicates can be combined to resemble typical database queries without including database specific code. Programs employing persistent predicates without database specific constructs can be used to access data stored in files or relational databases. Database specific combinators that allow for the generation of efficient database queries are introduced automatically by a program transformation.

In this thesis we introduce the database specific combinators and describe how they are translated into database queries. We also describe the program transformation that automatically augments a program with database specific code. Especially, the transformation automatically associates argument positions of persistent predicates with columns of database tables and generates database specific conditions from database independent ones. The conditions restrict the rows of the associated database tables. To restrict the requested columns, also projections are introduced.

Along with the transformation, useful programming techniques for meta-programming are introduced that allow for conveniently construct and decompose FlatCurry expressions. An inliner used to prepare a program for transformation is also described. The

inliner employs ideas found in [9], and a notion of descending argument positions is introduced to safely inline a class of directly recursive functions.

A prototype implementation of the presented approach has been compared 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 existing file based implementation. It is mandatory when the stored data is too voluminous to be held in main memory.

A. SQL Combinators

```
--- Expression pointing to the column of some table.
col :: Int -> SQLExp _

--- Constant value
val :: a -> SQLExp a

--- References a column by its name.
column :: String -> SQLExp _

--- Is expression the null value?
isNull :: SQLExp _ -> SQLExp Bool

--- Is expression not the null value?
isNotNull :: SQLExp _ -> SQLExp Bool

--- Is expression Nothing?
isNothing' :: SQLExp (Maybe _) -> SQLExp Bool

--- Is expression Just?
isJust' :: SQLExp _ -> SQLExp Bool

--- Is expression the empty list?
null' :: SQLExp [_] -> SQLExp Bool

--- Is argument element of the other?
--- The first argument has to be a literal or string,
--- the second has to be a database column.
elem' :: SQLExp a -> SQLExp [a] -> SQLExp Bool

--- Equality on expressions
(.==) :: SQLExp a -> SQLExp a -> SQLExp Bool

--- Less or equal
(.<=) :: SQLExp Int -> SQLExp Int -> SQLExp Bool
```

```
--- Greater or equal
(.>=) :: SQLExp Int -> SQLExp Int -> SQLExp Bool

--- Less than
(.<) :: SQLExp Int -> SQLExp Int -> SQLExp Bool

--- Greater than
(.>) :: SQLExp Int -> SQLExp Int -> SQLExp Bool

--- Boolean conjunction
(.&&) :: SQLExp Bool -> SQLExp Bool -> SQLExp Bool

--- Boolean disjunction
(.||) :: SQLExp Bool -> SQLExp Bool -> SQLExp Bool

--- Addition
(.+) :: SQLExp Int -> SQLExp Int -> SQLExp Int

--- Subtraction
(.-) :: SQLExp Int -> SQLExp Int -> SQLExp Int

--- Multiplication
(.*) :: SQLExp Int -> SQLExp Int -> SQLExp Int

--- Integer division
div' :: SQLExp Int -> SQLExp Int -> SQLExp Int

--- Match regular expression
(.=~) :: SQLExp String -> SQLExp String -> SQLExp Bool

--- Match SQL pattern
(.~~) :: SQLExp String -> SQLExp String -> SQLExp Bool

--- Negate boolean
not' :: SQLExp Bool -> SQLExp Bool

--- Invert integer
inv :: SQLExp Int -> SQLExp Int

--- Does first argument start with second?
--- Can only be applied if second argument is a constant!
startsWith' :: SQLExp String -> SQLExp String -> SQLExp Bool
```

```
--- Does first argument end with second?
--- Can only be applied if second argument is a constant!
endsWith' :: SQLExp String -> SQLExp String -> SQLExp Bool

--- Is first argument a substring of second?
substringOf' :: SQLExp String -> SQLExp String -> SQLExp Bool

--- Date comparison
before' :: SQLExp SQLDate -> SQLExp SQLDate -> SQLExp SQLDate

--- Year from date
year' :: SQLExp SQLDate -> SQLExp Int

--- Month from date
month' :: SQLExp SQLDate -> SQLExp Int

--- Day of month from date
day' :: SQLExp SQLDate -> SQLExp Int
```

B. Example: Transitive Closure

```
--- Shows how to mimic dynamic knowledge provided with getKnowledge.
--- Since getKnowledge can not (yet) be used together with database
--- predicates, the programmer has to query needed facts and to store
--- them as datatypes instead of calling getKnowledge. This example
--- shows that both access methods lead to similar programs.
---
--- @author Sebastian Fischer

import Dynamic
import AllSolutions

--- Computes the last element of a list.
last :: [a] -> a
last = head . reverse

--- A route from Kiel to Hamburg will be computed
data City = Kiel | Hamburg | Luebeck | Flensburg

connection :: (City,City) -> Dynamic
connection = dynamic

capital :: City -> Dynamic
capital = dynamic

--- Transitive closure of the connection predicate.
route :: (Dynamic -> Success) -> [City] -> Success
route known [c1,c2] = known (connection (c1,c2))
route known (c1:c2:cs)
  = known (connection (c1,c2)) & route known (c2:cs)

capital_route :: (Dynamic -> Success) -> [City] -> Success
capital_route known (c:cs)
  = route known (c:cs)
  & known (capital c)
  & known (capital (last cs))
```

```
init = assert $ foldr1 (<>)
  (map connection
   [(Flensburg,Kiel),(Kiel,Luebeck),(Luebeck,Hamburg),(Kiel,Hamburg)]
   ++ map capital [Kiel,Hamburg])

--- Prints routes from one capital to another.
main = do
  init
  known <- getKnowledge
  getAllSolutions (capital_route known) >>= mapIO_ print


connection_db :: (City,City) -> Dynamic
connection_db = persistent_db "db:currydb.connection"

capital_db :: City -> Dynamic
capital_db = persistent_db "db:currydb.capital"

data DynamicDB = Connection (City,City) | Capital City

--- Transitive closure of the connection predicate.
--- This cannot be tyranslated into one SQL query and is therefore
--- not constructed with Dynamic combinators.
route_db :: (DynamicDB -> Success) -> [City] -> Success
route_db known [c1,c2] = known (Connection (c1,c2))
route_db known (c1:c2:cs)
  = known (Connection (c1,c2)) & route_db known (c2:cs)

capital_route_db :: (DynamicDB -> Success) -> [City] -> Success
capital_route_db known (c:cs)
  = route_db known (c:cs)
  & known (Capital c)
  & known (Capital (last cs))

--- Prints routes from one capital to another.
--- The function known is defined explicitly to be used like
--- the function returned by getKnowledge.
main_db = do
  connections <- getDynamicSolutions connection_db
  capitals <- getDynamicSolutions capital_db
  let known (Connection c) = c =:= foldr1 (?) connections
      known (Capital c) = c =:= foldr1 (?) capitals
  getAllSolutions (capital_route_db known) >>= mapIO_ print
```

C. CiteSeer Database

```
--- Interface to a database storing scientific publications
--- obtained from http://citeseer.ist.psu.edu.
---
--- Persistent predicates are employed to access the database
--- and algebraic data types are defined to structure the rows
--- of the tables.
---
--- @author Sebastian Fischer

import Dynamic

data Record = Record Header Metadata

data Header = Header Identifier Datestamp SetSpec
type Identifier = Int
type Datestamp = SQLDate
type SetSpec = String

data Metadata = Citeseer
  (Title,[Author],Subject)
  (Maybe Description)
  ((Contributor,Publisher,Date),
   (Format,DCIdentifier,Source),
   (Language,(References,IsReferencedBy),Rights))

type Title = String

data Author = Author Name (Maybe Address) (Maybe Affiliation)
type Name = String
type Address = String
type Affiliation = String

type Subject = String
type Description = String
type Contributor = String
type Publisher = String
```

```
type Date = SQLDate
type Format = String
type DCIdentifier = String
type Source = String
type Language = String
type References = [Identifier]
type IsReferencedBy = [Identifier]
type Rights = String

recordDB :: Record -> Dynamic
recordDB = persistent_db "db:currydb.citeseer"

recordFile :: Record -> Dynamic
recordFile = persistent "file:/usr/local/dipl/citeseer/records"

data Pulication = Publication Int String [Int]

publication record (Publication nr title refs)
  = record (Record (Header nr X X)
     (Citeseer (title,X,X) X (X,X,(X,(refs,X),X))))

intMatch record title
  = publication record (Publication 10000 title X)

stringMatch record title
  = publication record (Publication X title X)
  |> "Curry" `substringOf` title

stringMatch' record title
  = publication record (Publication X title X)
  |> "Database" `substringOf` title

elemMatch record title
  = publication record (Publication X title refs)
  |> 3264 `elem` refs
 where
  refs free

countDynSols x = getDynamicSolutions x >>= print . length
```

Bibliography

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

[2] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 1970.

[3] J. Correas, J. M. Gómez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Model for Persistence in CLP Systems (And Two Useful Implementations). In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3507 in LNCS, pages 104–119. Springer-Verlag, June 2004.

[4] S. K. Das. *Deductive Databases and Logic Programming*. Addison Wesley, 1992.

[5] C. Draxler. *Accessing Relational and Higher Databases through Database Set Predicates in Logic Programming Languages*. PhD thesis, Department of Computer Science, Zurich University, 1991.

[6] M. Hanus, S. Antoy, 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/`, 2000.

[7] Michael Hanus. Dynamic Predicates in Functional Logic Programs. *Journal of Functional and Logic Programming*, 2004(5), December 2004.

[8] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.

[9] S. P. Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, July 2002.

[10] D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999. Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000).

[11] J. Shprentz. Persistent Storage of Python Objects in Relational Databases. In *Proceedings of the 6th International Python Conference*, 1997.

[12] P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

[13] ANSI X3.135-1992. Database Language SQL.