1 CurryPP: A Preprocessor for Curry Programs

The Curry preprocessor "`currypp`" implements various transformations on Curry source programs. It supports some experimental language extensions that might become part of the standard parser of Curry in some future version.

Currently, the Curry preprocessor supports the following extensions that will be described below in more detail:

Integrated code: This extension allows to integrate code written in some other language into Curry programs, like regular expressions, format specifications ("`printf`"), HTML and XML code.

Default rules: If this feature is used, one can add a default rule to operations defined in a Curry module. This provides a similar power than sequential rules but with a better operational behavior. The idea of default rules is described in [3].

Contracts: If this feature is used, the Curry preprocessor looks for contracts (i.e., specification, pre- and postconditions) occurring in a Curry module and adds them as assertions that are checked during the execution of the program. Currently, only strict assertion checking is supported which might change the operational behavior of the program. The idea and usage of contracts is described in [1].

1.1 Installation

The current implementation of Curry preprocessor is a package managed by the Curry Package Manager CPM. Thus, to install the newest version of `currypp`, use the following commands:

```
> cypm update
> cypm install currypp
```

This downloads the newest package, compiles it, and places the executable `currypp` into the directory `$HOME/.cpm/bin`. Hence one should add this directory to the path in order to use the Curry preprocessor as described below.

1.2 Basic Usage

In order to apply the preprocessor when loading a Curry source program into Curry, one has to add an option line at the beginning of the source program. For instance, in order to use default rules in a Curry program, one has to put the line

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=defaultrules #-}
```

at the beginning of the program. This option tells the Curry front end to process the Curry source program with the program `currypp` before actually parsing the source text.

The option "`defaultrules`" has to be replaced by "`contracts`" to enable dynamic contract checking. To support integrated code, one has to set the option "`foreigncode`" (which can also be combined with "`defaultrules`"). If one wants to see the result of the transformation, one can also set the option "`-o`". This has the effect that the transformed source program is stored in the file `Prog.curry.CURRYPP` if the name of the original program is `Prog.curry`.

For instance, in order to use integrated code and default rules in a module and store the transformed program, one has to put the line

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode --optF=defaultrules --optF=-o #-}
```

at the beginning of the program. If the options about the kind of preprocessing is omitted, all kinds of preprocessing are applied. Thus, the preprocessor directive

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp #-}
```

is equivalent to

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode --optF=defaultrules --optF=contracts #-}
```

1.3 Integrated Code

Integrated code is enclosed in at least two back ticks and ticks in a Curry program. The number of starting back ticks and ending ticks must always be identical. After the initial back ticks, there must be an identifier specifying the kind of integrated code, e.g., `regex` or `html` (see below). For instance, if one uses regular expressions (see below for more details), the following expressions are valid in source programs:

```
match ``regex (a|(bc*))+''
match ````regex aba*c''''
```

The Curry preprocessor transforms these code pieces into regular Curry expressions. For this purpose, the program containing this code must start with the preprocessing directive

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}
```

The next sections describe the currently supported foreign languages.

1.3.1 Regular Expressions

In order to match strings against regular expressions, i.e., to check whether a string is contained in the language generated by a regular expression, one can specify regular expression similar to POSIX. The foreign regular expression code must be marked by "`regex`". Since this code is transformed into operations of the Curry library `RegExp`, this library must be imported.

For instance, the following module defines a predicate to check whether a string is a valid identifier:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import RegExp

isID :: String → Bool
isID = match ``regex [a-zA-Z][a-zA-Z0-9_']*''
```

1.3.2 Format Specifications

In order to format numerical and other data as strings, one can specify the desired format with foreign code marked by "`format`". In this case, one can write a format specification, similarly to the `printf` statement of C, followed by a comma-separated list of arguments. This format specification is transformed into operations of the library `Data.Format` (of package `printf`) so that it must be imported. For instance, the following program defines an operation that formats a string, an integer (with leading sign and zeros), and a float with leading sign and precision 3:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import Data.Format

showSIF :: String → Int → Float → String
showSIF s i f = ``format "Name: %s | %+.5i | %+6.3f",s,i,f''

main = putStrLn $ showSIF "Curry" 42 3.14159
```

Thus, the execution of `main` will print the line

```
Name: Curry | +00042 | +3.142
```

Instead of "`format`", one can also write a format specification with `printf`. In this case, the formatted string is printed with `putStr`. Hence, we can rewrite our previous definitions as follows:

```
showSIF :: String → Int → Float → IO ()
showSIF s i f = ``printf "Name: %s | %+.5i | %+6.3f\n",s,i,f''

main = showSIF "Curry" 42 3.14159
```

1.3.3 HTML Code

The foreign language tag "`html`" introduces a notation for HTML expressions (see Curry library `HTML`) with the standard HTML syntax extended by a layout rule so that closing tags can be omitted. In order to include strings computed by Curry expressions into these HTML syntax, these Curry expressions must be enclosed in curly brackets. The following example program shows its use:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import HTML

htmlPage :: String → [HtmlExp]
htmlPage name = ``html
 <html>

  <head>
   <title>Simple Test

  <body>
   <h1>Hello {name}!</h1>
    <p>
```

```
      Bye!
     <p>Bye!
    <h2>{reverse name}
    Bye!''
```

If a Curry expression computes an HTML expression, i.e., it is of type `HtmlExp` instead of `String`, it can be integrated into the HTML syntax by double curly brackets. The following simple example, taken from [5], shows the use of this feature:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import HTML

main :: IO HtmlForm
main = return $ form "Question" $
         ``html
             Enter a string: {{textfield tref ""}}
             <hr>
             {{button "Reverse string"   revhandler}}
             {{button "Duplicate string" duphandler}}''

 where
  tref free

  revhandler env = return $ form "Answer"
    ``html <h1>Reversed input: {reverse (env tref)}''

  duphandler env = return $ form "Answer"
    ``html
       <h1>
         Duplicated input:
         {env tref ++ env tref}''
```

1.3.4 XML Expressions

The foreign language tag "`xml`" introduces a notation for XML expressions (see Curry library `XML`). The syntax is similar to the language tag "`html`", i.e., the use of the layout rule avoids closing tags and Curry expressions evaluating to strings (`String`) and XML expressions (`XmlExp`) can be included by enclosing them in curly and double curly brackets, respectively. The following example program shows its use:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=foreigncode #-}

import HTML

import XML

main :: IO ()
main = putStrLn $ showXmlDoc $ head ``xml
```

```
<contact>
 <entry>
  <phone>+49-431-8807271
  <name>Hanus
  <first>Michael
  <email>mh@informatik.uni-kiel.de
  <email>hanus@email.uni-kiel.de

 <entry>
  <name>Smith
  <first>Bill
  <phone>+1-987-742-9388
''
```

1.4 SQL Statements

The Curry preprocessor also supports SQL statements in their standard syntax as integrated code. In order to ensure a type-safe integration of SQL statements in Curry programs, SQL queries are type-checked in order to determine their result type and ensure that the entities used in the queries are type correct with the underlying relational database. For this purpose, SQL statements are integrated code require a specification of the database model in form of entity-relationship (ER) model. From this description, a set of Curry data types are generated which are used to represent entities in the Curry program (see Section 1.4.1). The Curry preprocessor uses this information to type check the SQL statements and replace them by type-safe access methods to the database. In the following, we sketch the use of SQL statements as integrated code. A detailed description of the ideas behind this technique can be found in [6]. Currently, only SQLite databases are supported.

1.4.1 ER Specifications

The structure of the data stored in underlying database must be described as an entity-relationship model. Such a description consists of

1. a list of entities where each entity has attributes,
2. a list of relationships between entities which have cardinality constraints that must be satisfied in each valid state of the database.

Entity-relationships models are often visualized as entity-relationship diagrams (ERDs). Figure 1 shows an ERD which we use in the following examples.

Instead of requiring the use of soem graphical ER modeling tool, ERDs must be specified in textual form as a Curry data term, see also [4]. In this representation, an ERD has a name, which is also used as the module name of the generated Curry code, lists of entities and relationships:

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

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

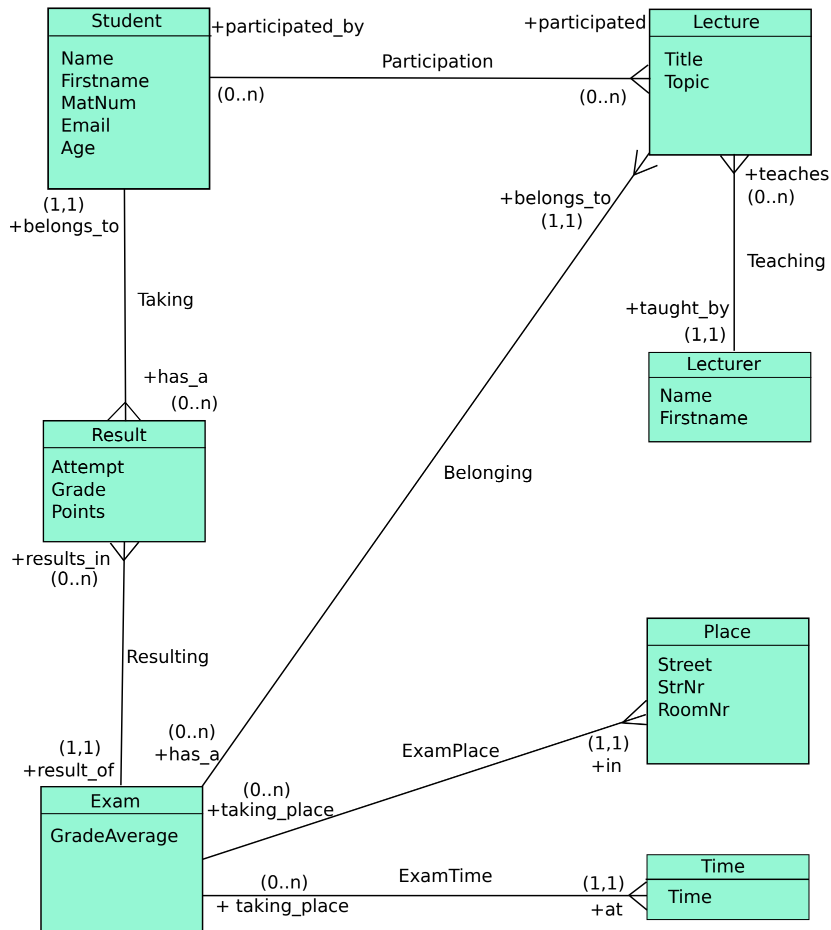

Figure 1: A simple entity-relationship diagram for university lectures [6]

```
data Entity = Entity String [Attribute]

data Attribute = Attribute String Domain Key Null

data Key = NoKey | PKey | Unique

type Null = Bool

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

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

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

```
data Relationship = Relationship String [REnd]

data REnd = REnd String String Cardinality

data Cardinality = Exactly Int | Between Int MaxValue

data MaxValue = Max Int | Infinite
```

The cardinality is either a fixed integer or a range between two integers (where `Infinite` as the upper bound represents an arbitrary cardinality). For instance, the simple-complex (1:n) relationship `Teaching` in Fig.1 can be represented by the term

```
Relationship "Teaching"
             [REnd "Lecturer" "taught_by" (Exactly 1),
              REnd "Lecture" "teaches"    (Between 0 Infinite)]
```

The Curry library `Database.ERD` contains the ER datatypes described above. Thus, the specification of the conceptual database model must be a data term of type `Database.ERD.ERD`. Figure 2 on (page 13) shows the complete ER data term specification corresponding to the ERD of Fig. 1.

Such a data term specification should be stored in Curry program file as an (exported!) top-level operation type `ERD`. If our example term is defined as a constant in the Curry program `UniERD.curry`, then one has to use the tool "`erd2curry`" to process the ER model so that it can be used in SQL statements. This tool is invoked with the parameter "`--cdbi`", the (preferably absolute) file name of the SQLite database, and the name of the Curry program containing the ER specification. If the SQLite database file does not exist, it will be initialized by the tool. In our example, we execute the following command (provided that the tool `erd2curry` is already installed:

```
> erd2curry --db `pwd`/Uni.db --cdbi UniERD.curry
```

This initializes the SQLite database `Uni.db` and performs the following steps:

1. The ER model is transformed into tables of a relational database, i.e., the relations of the ER model are either represented by adding foreign keys to entities (in case of (0/1:1) or (0/1:n) relations) or by new entities with the corresponding relations (in case of complex (n:m) relations).

2. A new Curry module `Uni_CDBI` is generated. It contains the definitions of entities and relationships as Curry data types. Since entities are uniquely identified via a database key, each

entity definition has, in addition to its attributes, this key as the first argument. For instance, the following definitions are generated for our university ERD (among many others):

```
data StudentID = StudentID Int

data Student = Student StudentID String String Int String Int

-- Representation of n:m relationship Participation:
data Participation = Participation StudentID LectureID
```

Note that the two typed foreign key columns (`StudentID`, `LectureID`) ensures a type-safe handling of foreign-key constraints. These entity descriptions are relevant for SQL queries since some queries (e.g., those that do not project on particular database columns) return lists of such entities. Moreover, the generated module contains useful getter and setter functions for each entity. Other generated operations, like entity description and definitions of their columns, are not relevant for the programming but only used for the translation of SQL statements.

3. Finally, an *info file* `Uni_SQLCODE.info` is created. It contains information about all entities, attributes and their types, and relationships. This file is used by the SQL parser and translator of the Curry preprocessor to type check the SQL statements and generate appropriate Curry library calls.

1.4.2 SQL Statements as Integrated Code

After specifying and processing the ER model of the database, one can write SQL statements in their standard syntax as integrated code (marked by the language tag "`sql`") in Curry programs. Since the SQL translator checks the correct use of these statements against the ER model, it needs access to the generated info file `Uni_SQLCODE.info`. This can be ensured in one of the following ways:

- The path to the info file is passed as a parameter prefixed by "`--model:`" to the Curry preprocessor, e.g., by the preprocessor directive

  ```
  {-# OPTIONS_CYMAKE -F --pgmF=currypp  --optF=--model:../Uni_SQLCode.info #-}
  ```

- The info file is placed in the same directory as the Curry source file to be processed or in one of its parent directories. The directories are searched from the directory of the source file up to its parent directories. If one of these directories contain more than one file with the name "`..._SQLCODE.info`", an error is reported.

After this preparation, one can write SQL statements in the Curry program. For instance, to retrieve all students from the database, one can define the following SQL query:

```
allStudents :: IO (SQLResult [Student])
allStudents = ``sql Select * From Student;''
```

Since the execution of database accesses might produce errors, the result of SQL statements is always of type "`SQLResult` τ", where `SQLResult` is a type synonym defined in the Curry library `Database.CDBI.Connection`:

```
type SQLResult a = Either DBError a
```

This library defines also an operation

```
fromSQLResult :: SQLResult a → a
```

which returns the retrieved database value or raises a run-time error. Hence, if one does not want to check the occurrence of database errors immediately, one can also define the above query as follows:

```
allStudents :: IO [Student]
allStudents = liftM fromSQLResult ``sql Select * From Student;''
```

In order to get more control on executing the SQL statement, one can add a star character after the language tag. In this case, the SQL statement is translated into a database action, i.e., into the type `DBAction` defined in the Curry library `Database.CDBI.Connection`:

```
allStudentsAction :: DBAction [Student]
allStudentsAction = ``sql* Select * From Student;''
```

Then one can put `allStudentsAction` inside a database transaction or combine it with other database actions (see `Database.CDBI.Connection` for operations for this purpose).

In order to select students with an age between 20 and 25, one can put a condition as usual:

```
youngStudents :: IO (SQLResult [Student])
youngStudents = ``sql Select * From Student
                      Where Age between 18 and 21;''
```

Usually, one wants to parameterize queries over some values computed by the context of the Curry program. Therefore, one can embed Curry expressions instead of concrete values in SQL statements by enclosing them in curly brackets:

```
studAgeBetween :: Int → Int → IO (SQLResult [Student])
studAgeBetween min max =
  ``sql Select * From Student
        Where Age between {min} and {max};''
```

Instead of retrieving complete entities (database tables), one can also project on some attributes (database columns) and one can also order them with the usual "`Order By`" clause:

```
studAgeBetween :: Int → Int → IO (SQLResult [(String,Int)])
studAgeBetween min max =
  ``sql Select Name, Age
        From Student Where Age between {min} and {max}
        Order By Name Desc;''
```

In addition to the usual SQL syntax, one can also write conditions on relationships between entities. For instance, the following code will be accepted:

```
studGoodGrades :: IO (SQLResult [(String, Float])
studGoodGrades = ``sql Select Distinct s.Name, r.Grade
                       From Student as s, Result as r
                       Where Satisfies s has_a r And r.Grade < 2.0;''
```

This query retrieves a list of pairs containing the names and grades of students having a grade better than `2.0`. This query is beyond pure SQL since it also includes a condition on the relation `has_a` specified in the ER model ("`Satisfies s has_a r`").

The complete SQL syntax supported by the Curry preprocessor is shown in Appendix A. More details about the implementation of this SQL translator can be found in [6, 7].

1.5 Default Rules

An alternative to sequential rules are default rules, i.e., these two options cannot be simultaneously used. Default rules are activated by the preprocessor option "`defaultrules`". In this case, one can add to each operation a default rule. A default rule for a function f is defined as a rule defining the operation "f`'default`" (this mechanism avoids any language extension for default rules). A default rule is applied only if no "standard" rule is applicable, either because the left-hand sides' pattern do not match or the conditions are not satisfiable. The idea and detailed semantics of default rules are described in [3].

Default rules are preferable over the sequential rule selection strategy since they have a better operational behavior. This is due to the fact that the test for the application of default rules is done with the same (sometimes optimal) strategy than the selection of standard rules. Moreover, default rules provide a similar power than sequential rules, i.e., they can be applied if the standard rules have complex (functional) patterns or complex conditions.

As a simple example, we show the implementation of the previous example for sequential rules with a default rule:

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=defaultrules #-}

mlookup key (_ ++ [(key,value)] ++ _) = Just value
mlookup'default _   _                 = Nothing
```

Default rules are often a good replacement for "negation as failure" used in logic programming. For instance, the following program defines a solution to the n-queens puzzle, where the default rule is useful since it is easier to characterize the unsafe positions of the queens on the chessboard (see the first rule of `safe`):

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=defaultrules #-}

import Combinatorial(permute)
import Integer(abs)

-- A placement is safe if two queens are not in a same diagonal:
safe (_++[x]++ys++[z]++_) | abs (x-z) == length ys + 1 = failed
safe'default xs = xs

-- A solution to the n-queens puzzle is a safe permutation:
queens :: Int  →  [Int]
queens n = safe (permute [1..n])
```

Important note: The implementation of default rules is based on set functions (implemented by the module `Control.SetFunctions`). Therefore, the package `setfunctions` should be installed as a dependency. This can easily done by executing

```
> cypm add setfunctions
```

before compiling a program containing default rules with the Curry preprocessor.

1.6 Contracts

Contracts are annotations in Curry program to specify the intended meaning and use of operations by other operations or predicates expressed in Curry. The idea of using contracts for the development of reliable software is discussed in [1]. The Curry preprocessor supports dynamic contract checking by transforming contracts, i.e., specifications and pre-/postconditions, into assertions that are checked during the execution of a program. If some contract is violated, the program terminates with an error.

The transformation of contracts into assertions is described in [1]. Note that only strict assertion checking is supported at the moment. Strict assertion checking might change the operational behavior of the program. The notation of contracts is defined in [1]. To transform such contracts into assertions, one has to use the option "`contracts`" for the preprocessor.

As a concrete example, consider an implementation of quicksort with a postcondition and a specification (where the code for `sorted` and `perm` is not shown here):

```
{-# OPTIONS_CYMAKE -F --pgmF=currypp --optF=contracts #-}

...

-- Trivial precondition:
sort'pre xs = length xs >= 0

-- Postcondition: input and output lists should have the same length
sort'post xs ys = length xs == length ys

-- Specification:
-- A correct result is a permutation of the input which is sorted.
sort'spec :: [Int]  → [Int]
sort'spec xs | ys == perm xs && sorted ys = ys  where ys free

-- A buggy implementation of quicksort:
sort :: [Int]  → [Int]
sort []     = []
sort (x:xs) = sort (filter (<x) xs) ++ [x] ++ sort (filter (>x) xs)
```

If this program is executed, the generated assertions report a contract violation for some inputs:

```
Quicksort> sort [3,1,4,2,1]
Postcondition of 'sort' (module Quicksort, line 27) violated for:
[1,2,1]  → [1,2]

ERROR: Execution aborted due to contract violation!
```

Important note: The implementation of default rules is based on set functions (implemented by the module `Control.SetFunctions`) and the auxiliary package to check contracts at run time. Therefore, the packages `setfunctions` and `contracts` should be installed as dependencies. This can

easily done by executing

```
> cypm add setfunctions
> cypm add contracts
```

before compiling a program containing contracts with the Curry preprocessor.

```
ERD "Uni"
  [Entity "Student"
          [Attribute "Name" (StringDom Nothing) NoKey False,
           Attribute "Firstname" (StringDom Nothing) NoKey False,
           Attribute "MatNum" (IntDom Nothing) Unique False,
           Attribute "Email" (StringDom Nothing) Unique False,
           Attribute "Age" (IntDom Nothing) NoKey True],
   Entity "Lecture"
          [Attribute "Title" (StringDom Nothing) NoKey False,
           Attribute "Topic" (StringDom Nothing) NoKey True],
   Entity "Lecturer"
          [Attribute "Name" (StringDom Nothing) NoKey False,
           Attribute "Firstname" (StringDom Nothing) NoKey False],
   Entity "Place"
          [Attribute "Street" (StringDom Nothing) NoKey False,
           Attribute "StrNr" (IntDom Nothing) NoKey False,
           Attribute "RoomNr" (IntDom Nothing) NoKey False],
   Entity "Time"
          [Attribute "Time" (DateDom Nothing) Unique False],
   Entity "Exam"
          [Attribute "GradeAverage" (FloatDom Nothing) NoKey True],
   Entity "Result"
          [Attribute "Attempt" (IntDom Nothing) NoKey False,
           Attribute "Grade" (FloatDom Nothing) NoKey True,
           Attribute "Points" (IntDom Nothing) NoKey True]]
  [Relationship "Teaching"
                [REnd "Lecturer" "taught_by" (Exactly 1),
                 REnd "Lecture" "teaches" (Between 0 Infinite)],
   Relationship "Participation"
                [REnd "Student" "participated_by" (Between 0 Infinite),
                 REnd "Lecture" "participates" (Between 0 Infinite)],
   Relationship "Taking"
                [REnd "Result" "has_a" (Between 0 Infinite),
                 REnd "Student" "belongs_to" (Exactly 1)],
   Relationship "Resulting"
                [REnd "Exam" "result_of" (Exactly 1),
                 REnd "Result" "results_in" (Between 0 Infinite)],
   Relationship "Belonging"
                [REnd "Exam" "has_a" (Between 0 Infinite),
                 REnd "Lecture" "belongs_to" (Exactly 1)],
   Relationship "ExamDate"
                [REnd "Exam" "taking_place" (Between 0 Infinite),
                 REnd "Time" "at" (Exactly 1)],
   Relationship "ExamPlace"
                [REnd "Exam" "taking_place" (Between 0 Infinite),
                 REnd "Place" "in" (Exactly 1)]]
```

Figure 2: The ER data term specification of Fig. 1

A SQL Syntax Supported by CurryPP

This section contains a grammar in EBNF which specifies the SQL syntax recognized by the Curry preprocessor in integrated SQL code (see Sect. 1.4). The grammar satisfies the LL(1) property and is influenced by the SQLite dialect.[1]

```
--------------type of statements--------------------------------

 statement ::= queryStatement | transactionStatement
 queryStatement ::= ( deleteStatement
                    | insertStatement
                    | selectStatement
                    | updateStatement )
                    ';'

 ------------- transaction ------------------------------------

 transactionStatement ::= (BEGIN
                          |IN TRANSACTION '(' queryStatement
                                            { queryStatement }')'
                          |COMMIT
                          |ROLLBACK ) ';'

-------------- delete -----------------------------------------

 deleteStatement ::= DELETE FROM tableSpecification
                                [ WHERE condition ]

 -------------insert -------------------------------------------

 insertStatement ::= INSERT INTO tableSpecification
                                insertSpecification

 insertSpecification ::= ['(' columnNameList ')' ] valuesClause

 valuesClause ::= VALUES valueList

  ------------update--------------------------------------------

 updateStatement ::= UPDATE tableSpecification
                    SET (columnAssignment {',' columnAssignment}
                                         [ WHERE condition ]
                        | embeddedCurryExpression )

 columnAssignment ::= columnName '=' literal


 -------------select statement ---------------------------------
```

[1] https://sqlite.org/lang.html

```
selectStatement ::= selectHead { setOperator selectHead }
                              [ orderByClause ]
                              [ limitClause ]
selectHead ::= selectClause fromClause
                           [ WHERE condition ]
                           [ groupByClause [ havingClause ]]

setOperator ::= UNION | INTERSECT | EXCEPT

selectClause ::= SELECT [( DISTINCT | ALL )]
                        ( selectElementList | '*' )

selectElementList ::=  selectElement { ',' selectElement }

selectElement ::= [ tableIdentifier'.' ] columnName
                 | aggregation
                 | caseExpression

aggregation ::= function '(' [ DISTINCT ] columnReference ')'

caseExpression ::= CASE WHEN condition THEN operand
                                       ELSE operand END

function ::= COUNT | MIN | MAX | AVG | SUM

fromClause ::= FROM tableReference  { ',' tableReference }

groupByClause ::= GROUP BY columnList

havingClause ::= HAVING conditionWithAggregation

orderByClause ::= ORDER BY columnReference [ sortDirection ]
                          {',' columnReference
                               [ sortDirection ] }

sortDirection ::= ASC | DESC

limitClause = LIMIT integerExpression

-------------common elements-----------------------------------

columnList ::=  columnReference  { ',' columnReference }

columnReference ::= [ tableIdentifier'.' ] columnName

columnNameList ::= columnName  { ',' columnName}

tableReference ::=  tableSpecification [ AS tablePseudonym ]
```

```
                                       [ joinSpecification ]
tableSpecification ::= tableName

condition ::=   operand operatorExpression
                        [logicalOperator condition]
              | EXISTS subquery [logicalOperator condition]
              | NOT condition
              | '(' condition  ')'
              | satConstraint [logicalOperator condition]

operand ::=  columnReference
            | literal


subquery ::= '(' selectStatement ')'

operatorExpression ::=  IS NULL
                      | NOT NULL
                      | binaryOperator operand
                      | IN setSpecification
                      | BETWEEN operand operand
                      | LIKE quotes pattern quotes

setSpecification ::=  literalList

binaryOperator ::= '>'| '<' | '>=' | '<=' | '=' | '!='

logicalOperator ::= AND | OR

conditionWithAggregation ::=
        aggregation [logicalOperator disaggregation]
      | '(' conditionWithAggregation  ')'
      | operand operatorExpression
               [logicalOperator conditionWithAggregation]
      | NOT conditionWithAggregation
      | EXISTS subquery
               [logicalOperator conditionWithAggregation]
      | satConstraint
               [logicalOperator conditionWithAggregation]

aggregation ::= function '('(ALL | DISTINCT) columnReference')'
                          binaryOperator
                          operand

satConstraint ::= SATISFIES tablePseudonym
                            relation
                            tablePseudonym

joinSpecification ::=  joinType tableSpecification
```

```
                              [ AS tablePseudonym ]
                              [ joinCondition ]
                              [ joinSpecification ]

joinType ::= CROSS JOIN | INNER JOIN

joinCondition ::= ON condition

-------------identifier and datatypes------------------------

valueList ::= ( embeddedCurryExpression | literalList )
               {',' ( embeddedCurryExpression | literalList )}

literalList ::= '(' literal  { ',' literal }  ')'

literal ::=   numericalLiteral
            | quotes alphaNumericalLiteral quotes
            | dateLiteral
            | booleanLiteral
            | embeddedCurryExpression
            | NULL

numericalLiteral ::=  integerExpression
                     |floatExpression

integerExpression ::= [ - ] digit { digit }

floatExpression := [ - ] digit { digit } '.' digit { digit }

alphaNumericalLiteral ::=  character { character }
character ::= digit | letter

dateLiteral ::=  year ':' month ':' day ':'
                 hours ':' minutes ':' seconds

month ::= digit digit
day ::= digit digit
hours ::= digit digit
minutes ::= digit digit
seconds ::= digit digit
year ::= digit digit digit digit

booleanLiteral ::= TRUE | FALSE

embeddedCurryExpression ::= '{' curryExpression '}'

pattern ::= ( character | specialCharacter )
             {( character | specialCharacter )}
specialCharacter ::= '%' | '_'
```

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

letter ::= (a...z) | (A...Z)

tableIdentifier ::= tablePseudonym | tableName
columnName ::= letter [alphanumericalLiteral]
tableName ::= letter [alphanumericalLiteral]
tablePseudonym ::= letter
relation ::= letter [[alphanumericalLiteral] | '_' ]
quotes ::= ('"'|''')
```

References

[1] S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012.

[2] S. Antoy and M. Hanus. Curry without Success. In *Proc. of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)*, volume 1335 of *CEUR Workshop Proceedings*, pages 140–154. CEUR-WS.org, 2014.

[3] S. Antoy and M. Hanus. Default rules for Curry. In *Proc. of the 18th International Symposium on Practical Aspects of Declarative Languages (PADL 2016)*, pages 65–82. Springer LNCS 9585, 2016.

[4] B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.

[5] M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[6] M. Hanus and J. Krone. A typeful integration of SQL into Curry. In *Proceedings of the 24th International Workshop on Functional and (Constraint) Logic Programming*, volume 234 of *Electronic Proceedings in Theoretical Computer Science*, pages 104–119. Open Publishing Association, 2017.

[7] J. Krone. Integration of SQL into Curry. Master's thesis, University of Kiel, 2015.