relationMCQDSL
Project Report
CS 583

Fariba Khan
School of EECS
Oregon State University

June 12, 2020

1 Overview

Different field of studies, such as biology, sociology, mathematics and even computer science often times has rich classification or some other dependence relationships among objects. Creating multiple choice questions (MCQ) is a tedious job. It takes substantial amount of time for instructors, especially to find correct and incorrect choices from domain which entails structuring subset of knowledge either in head or on paper to look up for options. This process is repeated each time a quiz is created. Motivation behind this work to create a dedicated quiz generator was this observation of loss of work and time consuming manual look up. With the hope of making the process less demanding and possibly fun, here, I present a DSL, relationMCQDSL that provides a structure for describing domain knowledge once and then use it repeatedly for quiz generation with the help of automated look up functions.

Similar existing tools include UML that can be used to describe object relationships, behavior or interactions in a system. Prolog programs also describe relations. But none of them provides a mechanism for quiz question generation exploiting these relationships.

2 Functionalities and Outcome

relationMCQDSL lets domain expert to represent knowledge such as relationship among domain entities through mathematical relation, provides compositional operations on relations to combine or derive complicated relationships from basic ones and gives constructions for quiz creation with MCQs. Functionalities for quiz generation include structure to define multiple choice questions over described relations, set scores and explanation for each choice of a question and combine questions to create a quiz. DSL also provides function to find correct and incorrect choices i.e. answers and distractors to defined questions automatically.

The outcome of executing a program written in DSL should be a MCQ quiz (set of multiple choice questions) with embedded scoring system and explanation that can be printed in any domain or can be used to simulate an interactive quiz. A function `executeMCQ` has been provided as part of DSL to accomplish the later.

3 Users

Domain experts burdened with creating MCQ questions and whose fields or part of the fields can be described through relationships/ dependencies are the active users. Students and learners in those domains are also user of the DSL as they are the ones who take the quizzes, thus get affected by its quality.

4 Types, Type classes, and Functions

The implementation is structured into three main components - Relation (BinRelationV3.hs), Quiz Generator (QuesGeneratorV3.hs) and Quiz Executor (ExecuteQuiz.hs). Relation and Quiz genrator files contain respective data type, smart constructors, combinators and functions that work on them (information extraction or manipulation). Quiz executor holds the functions to run quiz interactively in an command-line interface.

Also, an example (SimpleExample.hs) has been included in the project to illustrate use of DSL. It contains representation of three domains with Relation and formation of a single quiz combining questions of different types from example domains. Main function executes any given quiz created in DSL with the Quiz Executor function.

4.1 Basic Objects

The overview of the basic objects in relationMCQDSL is listed below.

- `Relation a b` is the basic object to encode relationship among domain objects in DSL. Recall, Relation from domain A to range B is a subset of Cartesian product of A, B. Components of `Relation` is as illustrated below.

```
data Relation a b  = Relation
{ elements       :: S.Set (a, b)
, falseElements :: S.Set (a, b)
, relatedBy     :: String
, dname         :: String
, rname         :: String
}
```

  `elements` holds all pairs that belong to the relation. `falseElements` holds some pairs that doesn't belong to the described relationship. It can be useful to provide more options for distractors. `relatedBy` , `dname` and `rname` captures the names of relationship itself and domain and range entities respectively. Having non meta-level names stored inside relation has some useful application in our context.

  Names once set can be used to create bulk of questions from a `Relation` without having to set question phrase for each question. For example, from a `Relation` containing all (father, son) pairs, with domain name set as `father` and range name, as `son` and relationship name, as `father-son`, phrases such as `has father-son relationship with`, `is father of` and `is son of` can be instantiated with multiple pairs from `Relation` elements, and questions can be asked about those phrases. Functions have been provided in DSL to create such phrase strings from names. Relation combinators work on the names as well to combine them in a meaningful way with respect to the operations.

- `Statement a` describes a question statement from some `Relation`. Consider, for instance, `object a` has `R` relationship with `object b`. A question can be made by replacing right element `object b` with blank from the statement and asking for valid answer for the blank. Generated question would look like: `object a` has `R` relationship with ____. Similarly, `object a` can be omitted to get a question for left element. Therefore, `Statement` has type `data Statement a = S K a String` where String holds the phrase to ask the question. `K` is used for printing question and can have value L or R indicating whether value `a` should be placed on the left (K = L) or right (K = R) side of the string. A blank will be printed on the other side. Using above `father-son` example, for a pair element (Mat, Ron), question statements can be like

```
S L Mat "is father of" | printed as: Mat is father of ___?
S R Ron "is father of" | printed as: ___ is father of Ron?
S L Ron "is son of"    | printed as: Ron is son    of ___?
```

- `Options a v` is the object to create choices for multiple choice questions and has following type.

  ```
  data Options a v where
  Item  :: a -> v -> Double -> Options a v
  ```

  `a` is the option value (an answer or a distractor), `v` holds an explanation or description for the option and Double holds a score. It allows to embed a scoring and feedback system within the question.

- Finally, `data MCQ :: NumA -> Statement a -> [Options b v] -> Question a b` is the object to create a Multiple Choice Question which includes a `Statement a`, list of `Options a v` and an indication, `data NumA = SA | MA` to whether examinee should be asked to select one choice or all choices that apply as answer.

4.2 Operators and Combinators

Combinators for the `Relation` object are `union`, `intersection`, `difference`, `composition` and `product` of two mathematical relations.

```
union        :: Relation a b -> Relation a b -> Relation a b
intersection :: Relation a b -> Relation a b -> Relation a b
difference   :: Relation a b -> Relation a b -> Relation a b
compose      :: Relation a b -> Relation b c -> Relation a c
prod         :: Relation a b -> Relation c d -> Relation (a, b) (c, d)
```

`union`, `intersection` and `difference` combines the set of `elements` and `falseElements` of the two given relations with set union, set intersection and set difference operations. New names are generated as follows:

```
relatedBy (r ‘union‘ s)        =  relatedBy r ++ "-or-"  ++ relatedBy s
relatedBy (r ‘intersection‘ s) =  relatedBy r ++ "-and-" ++ relatedBy s
relatedBy (r ‘difference‘ s)   =  relatedBy r ++ "-not-" ++ relatedBy s
-- dname and rname are kept same as left Relation r.
```

`composition` of two relations r of type `Relation a b` and s of type `Relation b c` creates a new higher order relation of type `Relation a c` such that

```
-- element composition
elements      (r ‘compose‘ s) = {(a, c) | (a, b) <- elements r,      (b, c) <- elements s )}
falseElements (r ‘compose‘ s) = {(a, c) | (a, b) <- falseElements r, (b, c) <- falseElements s )}
-- Name composition
relatedBy     (r ‘compose‘ s)  =  relatedBy r ++ "-to-" ++ relatedBy s
dname         (r ‘compose‘ s)  =  dname r
rname         (r ‘compose‘ s)  =  rname s
```

`product` also produces a higher order relation by computing Cartesian product of the `elements` (and `falseElements`) of two relations. For all, `(a, b)` in `Relation` r and `(c, d)` in `Relation` s, product of r and s, say rs contains all `((a, b) , (c, d))` pairs. Names are concatenated as, relatedBy rs = (relatedBy r ++ ”–” ++ relatedBy s), dname rs = (dname r ++’,’++ rname r), rname rs = (dname s ++’,’++ rname s).

Multiple Choice Question (MCQ) type and their combinators are provided by the following GADT definition.

```
data Question a b where
MCQ     :: NumA -> Statement a  -> [Options b v]  -> Question a b
(:++:)  :: Question a b -> Question c d  -> Question (a, c) (b, d)
Quiz    :: String       -> Question a b  -> Question a b
```

Constructor `MCQ` has been described above. `(:++:)` combines two questions of different types. Type of the resulted question is the pair of the types of constituting questions. Quiz can be used to group a set of questions under some title.

4.3 Functions

Aside from basic objects and combinators for the core DSL, several smart constructors and utility functions have been provided within the DSL. Few of them are discussed here that are important and interesting enough to be mentioned. An example DSL program is generated as part of the discussion to illustrate each function’s utility and output.

Relation has three basic smart constructors, `empty` creates an empty Relation, `singleton` creates a singleton Relation from a single pair and `fromList` takes a list of pairs and creates a relation from it with names set to NULL.

```
empty :: Relation a b
singleton :: a -> b -> Relation a b
fromList :: (Ord a, Ord b) => [(a, b)] -> Relation a b
```

`fromList` has a named version `fromListWithNames` that sets names from arguments and a Full Relation from domain, range version, `fromDomRanFullRWNames` that takes domain and range as lists and create a Relation from their Cartesian product.

For example, if we have a list of mother-child pairs,

```
motherchildL = [("Alice", "Bob"  ),
                ("Alice", "Rachel"),
                ("Jane" , "Alice" ),
                ("April", "Jane"  )]
```

we can create a relation by calling smart constructor `fromList` with the list.

```
motherchildR = fromList motherchildL
```

Similarly, a father-child relation can be constructed which uses the named version of the constructor.

```
fatherchildL = [("Patrick", "Bob"    ),
                ("Patrick", "Rachel" ),
                ("Matthew", "Patrick"),
                ("Leonard", "Matthew")]

fatherchildR = fromListWithNames "father-child" "father" "child" fatherchildL
```

`motherchildR` and `fatherchildR` can be combined with combinator ‘union‘ to get parent-child relationship.

```
parentchildR' = motherchildR `union` fatherchildR
parentchildR  = setNames "parent-child" "parent" "child" parentchildR'
```

Higher order relations such as grandparent-child, greatgrandparent-child can be derived with ‘compose‘ operation on parentchildR relation.

```
grandParentChildR      = parentchildR `compose` parentchildR
greatgrandParentChildR = grandParentChildR `compose` parentchildR
```

Thus, only primitive relations require tedious work of finding all pairs.

Several helper functions have been provided to extract information from Relation such as

```
getDomain   ::  Relation a b -> [a]
getRange    ::  Relation a b -> [b]
getDomainOf ::  b   -> Relation a b -> [a]
getRangeOf  ::  a   -> Relation a b -> [b]
getNthDomain::  Int -> Relation a b -> a
getNthRange::   Int -> Relation a b -> b
```

`getDomainOf` takes an element of range type and finds all domain elements that are related to it. `getRangeOf` does same with an element of domain type. These functions are important as they are used to get answers and distractors for MCQs. `getNthDomain` and `getNthRange` give nth element from domain or range of a Relation. Both of them have a safe version that checks if n is within respective size.

Quiz generator provides smart constructors and functions to create or get different parts of a question such as `Statement` along with its value and question phrase; correct and incorrect choices for a MCQ; list of `Options` from combination of choices, explanations, scores.

`getQStringRelatedBy`, `getQStringDname` and `getQStringRname` create question phrase respectively from relation, domain and range name of a `Relation`. All of them have type `Relation a b -> String`. For example,

```
qsD = getQStringDname fatherchildR -- >>> "is father of"
qsR = getQStringRname fatherchildR -- >>> "is child of"
```

Smart constructors for `Statement a` captures all possible combinations of statement created from a `Relation`. If we recall it's type data Statement a = S K a String, statement can have K value set to L or R indicating value `a`'s position in printed question and value `a` containing a domain or range element of a Relation. String can be created from relationship, domain or range name using above functions. Below are the two smart constructors.

```
-- | get a statement with K, nth domain element and string set with provided function
getDmStatement ::  K -> Int -> (Relation a b -> String) -> Relation a b -> Statement a
-- | get a statement with K, nth range element and string set with provided function
getRnStatement ::  K -> Int -> (Relation a b -> String) -> Relation a b -> Statement b
```

Calling `getDmStatement` with running example,

```
s1 = getDmStatement L 0 getQStringDname fatherchildR
-- >>> S L "Leonard" "is father of"
-- | printed as: Leonard is father of ______?
```

Now, say we want to pose the same question differently as ______ `is child of Leonard?`. This can be done simply by switching K value and name generation function.

```
s2 = getDmStatement R 0 getQStringRname fatherchildR
-- >>> S R "Leonard" "is child of"
-- | printed as: ______ is child of Leonard?
```

Similarly, statement with a range value can be generated as below,

```
s3 = getRnStatement R 0 getQStringDname fatherchildR
-- >>> S R "Bob" "is father of"
-- | printed as: _______ is father of Bob?
```

Correct and incorrect choices are generated automatically for a Statement with following functions. `getDmAnswers` and `getDmDistractors` generate all correct and incorrect options i.e. answers and distractors for a statement containing a domain value. `getRnAnswers` and `getRnDistractors` generate all answers and distractors for a statement containing a range value.`getOptsFromAnsDists` creates list of options by putting together answers, distractors with respective explanations and scores.

```
ans1  = getDmAnswers s1 fatherchildR -- as s1 contains a domain element
dist1 = getDmDistractors s1 fatherchildR

ans3  = getRnAnswers s3 fatherchildR -- as s3 contains a range element
dist3 = getRnDistractors s3 fatherchildR
```

Quiz questions can be generated from above values as,

```
q1 = MCQ SA s1 (getOptsFromAnsDists 1 ans1  ["Correct"]                [1.0]
                                    2 dist1 ["Incorrect", "Incorrect"] [-0.25, -0.25])

-- which basically says create an single answer (SA) MCQ, with statement s1 and taking
-- 1 correct option from answers setting its explanation as Correct, score to 1.0 and
-- 2 incorrect options from distractors.

q3 = MCQ MA s3 (getOptsFromAnsDists 1 ans3  [True]          [1.0]
                                    2 dist3 [False, False]  [0.0, 0.0])

-- q1 qnd q3 can combined with (:++:)
q13  = q1 :++: q3
quiz = Quiz "Family Tree Questions" q13
```

Finally, Quiz executor contains `execQuiz` function that runs a given quiz created in DSL in an interactive manner.

```
execQuiz :: Question a b -> StateT Int IO (Writer String Double)
```

- State monad `StateT Int` is used to display questions with numbers.
- Writer monad `Writer String Double` is used to log explanation for user's selected options and accumulate score.
- Several funtions are called from it to accomplish different aspects of interactive quiz simulation.
  - `getAnswers` is called to display questions and collect and filter responses (described below).
    - displays questions with randomly shuffled options
    - collects user responses.
    - filters user responses that are not numbers or not within the number of options. For instance, if four options are provided, only numbers [1...4] are deemed as valid response. A response as ”1 p2” would be interpreted as ”1 2”. No input would be handled as No answers. Also, if question indicates that only one answer is allowed (`NumA = SA`), anything after first valid response will be discarded as well.
  - `checkM` is called to check responses against shuffled options.
  - `printCollect` is called to log feedback for selected options and accumulate score.

5 Design Decisions and Evolution

`Relation a b` and `MCQ` types define semantic domain of the object language. Syntax of the language can be extended with functions that creates Relation and MCQ values. Several smart constructors have already been provided within DSL.

GADTs is used for `Question a b` to explicitly mention type of combined questions so that ill-typed questions in object language are prevented through host language, Haskell's type system.

`Statement a`'s type initially was `data Statement a = L a String | R a String` which required handling unnecessary pattern matching cases in function and more variation of smart constructors. Current data type `data Statement a = S K a String; data K = L | R` enables functions to ignore K values when not required or to take its value as arguments in constructors.

Explanations in `Options` could just be stored as String, making current type to `data Options a v = Item a v Double` more concrete as `data Options a = Item a String Double`. Handling error cases such as explanations not set would entail just handling NULL string. But I decided to kept it open for any unforseen future use. And, it always can be set to String or user can decide to make `v` a Maybe value to be able to set it to Nothing.

`executeQuiz` function went through several changes of its type and few iteration of refactoring. I started with following types for Options and executeQuiz,

```
data Options a v = Item a v
execQuiz :: execQuiz :: Question a b v -> IO v
```

Options only had one associated value, instead of separate explanation and score. Initially, `execQuiz` only displayed the value of v's for user selected options. It also required value v to be part of Question type to be able correctly create v for combined questions. In next iteration, I decided to separate explanation and score,

```
data Options a v = Item a v Double
execQuiz :: Question a b -> IO (Writer String Double)
```

To log explanation for selected options and accumulate score as we take response, I used a writer monad. As at this point, explanation are logged to show as feedback to user, converting them to String made sense. It simplified combined questions type as well, from `(:++:):: Question a b v1 -> Question c d v2 -> Question (a, c) (b, d) (v1, v2)` to `(:++:):: Question a b -> Question c d -> Question (a, c) (b, d)` . However, up until this point, I was not using numbers for displayed questions. Showing explanation at the end for responses without question numbers seemed less comprehensible, thus I introduced State monad with monad transformer to combine it with IO effects for numbering question as they are being displayed.

```
data Options a v = Item a v Double
execQuiz :: execQuiz :: Question a b -> StateT Int IO (Writer String Double)
```

`execQuiz` also went through refactoring, specially pattern matching on MCQ has been factored out. As it's only required for printing questions, now only a dedicated print function pattern matches on MCQ.

6 Future Work

The use of `falseElemnts` has not been fully investigated in this work. The original motivation behind its inclusion was to capture not so obvious untrue values for misleading incorrect choices. This can be done as future extension to this work.