Static checking of domain constraints in applications interacting with relational database by means of dependently-typed lambda calculus

Maxim Aleksandrovich Krivchikov

Ph.D., senior researcher, Lomonosov Moscow State University
maxim.krivchikov@gmail.com

Evgeny Vladimirovich Shulgin

graduate, Lomonosov Moscow State University
shulginev@yandex.ru

1 Introduction

In the modern software development practice, almost every information system uses a database management system (DBMS) to store data. The most widespread database model nowadays is the relational model, which describes data as the tuples of attribute values. This model is implemented by relational database management systems. One of the aspects of data modeling in relational DBMS is the consistency constraints.

Consistency constraints can be represented either as predicates over the attributes of a single tuple, or as uniqueness requirements of the subset of attributes in the whole relation, as a requirement of the existence of tuple with specified attributes in a different relation, or as a subroutine call before appending the tuple to the relation. Such subroutine call in existing relational DBMS may perform arbitrary database queries. Requests for data modification (appending or removal of a tuple or updating the attribute values) which result in a violation of consistency constraints are rejected by DBMS.

The common design models for information systems interacting with DBMS are three-tier or client-server architectures. First tier (data tier) corresponds to the database. Second tier (logic tier or client) is usually implemented as an application server or a separate application written in a general purpose programming language (e.g., Java, C#, Python). Interaction with the database is implemented in the application by means of SQL language or a software library which constructs SQL queries.

The application code must be consistent with the data model, which means that all possible cases of consistency constraint violation must be handled in the application. During the software lifecycle data model and constraints are usually modified to reflect the changes in requirements. Therefore, the consistency constraints handling in the application must be controlled continuously. The present paper describes the research-in-progress of the relational database model in terms of dependently-typed lambda calculus. The model can represent the consistency constraints on the relational

database and statically check that application code respects the consistency constraints.

2 Background

We will start the discussion by an example from ISTINA system developed in Lomonosov Moscow State University. We will discuss the domain constraints in "Dissertation Councils" subsystem which is concerned with the informational support of public doctoral theses defense. The data model of the simplified "Dissertation Councils" subsystem in example consists of the following relations:

1. Researcher profile ("Worker" relation) which in present paper contains a single attribute "Worker.id" which uniquely identifies the profile of the researcher in the system (usually called "primary key").
2. Dissertation Council ("DissertationCouncil" relation) with the following attributes: DissertationCouncil.id – the primary key; DissertationCouncil.number – the unique sequence of letters and numbers separated by space and period characters, identifying the dissertation council in official documents.
3. Record of membership of the scientist in the dissertation council ("DissertationCouncilMembership" relation) with the following attributes: DissertationCouncilMembership.id – the primary key; DissertationCouncilMembeship.worker – the reference to researcher profile (a "foreign key"); DissertationCouncilMembership.council – the reference to the dissertation council (a "foreign key"); DissertationCouncilMembership.startdate – the effective date of entry of the scientist to the dissertation council; DissertationCouncilMembership.enddate – the effective date of resignation of the scientist from the dissertation council.

The consistency constraints on these relations include:

- primary keys uniqueness (by definition of primary key);
- foreign key existence (by definition of foreign key);
- dissertation council number uniqueness (domain requirement);
- a scientist can simultaneously be a member of no more than two dissertation councils in Lomonosov Moscow State University (domain requirement).

Modern software development frameworks include the database abstraction libraries. With such libraries, an entry may be inserted into the database by means of the function call of the library. For example, in Django framework for Python programming language the insertion of new dissertation council membership record is represented by the following line of code:

```
DissertationCouncilMembership.objects.create(worker=w,
council=c, startdate=datetime.now())
```

If the previously mentioned constraints are defined in DBMS, the evaluation of this statement may result in an error (exception). Such exceptions are usually handled in

the software code. Statically typed programming languages (for example, in Java programming language) usually provide tools for checking that all possible types of exceptions which can be raised inside the function are handled. Nevertheless, exceptions produced by DBMS often are rather generic (e.g. “integrity constraint (SOME_POSSIBLY_MEANINGFUL_NAME) violated – child record found”) and hence hard to process. Also, the handling of all possible exceptions cannot ensure that all data constraints are handled in exception handlers (which results in undescriptive error messages). More importantly, it is not possible to ensure that some fragment of code will not violate any consistency constraints due to performed checks of input data.

The model presented in this paper allows representing the results of constraint check as so-called “witness object” to prove that the insertion of a new row will not result in constraint violation. The type of such witness object binds it with the specific data set for which the checks are performed and with the details of checks. A function which inserts the record into the database with static constraint control accepts such proof as an additional parameter:

```
P = check_no_more_than_one_membership(worker=w, council=c,
startdate=datetime.now())
DissertationCouncilMember-
ship.objects.create(worker=w, council=c,
startdate=datetime.now(), proof=P)
```

The model is based on dependently-typed lambda calculus and implemented in Agda proof assistant. The model itself does not use any features specific to Agda except type inference in examples. Presumably, the model can be implemented with minimal modifications in any dependently-typed language based on the Calculus of Constructions supporting mutually inductive type definitions.

3 Related work

The existing work on relational DBMS specification in dependently-typed programming languages is concerned primarily with the task of verified DBMS implementation. For example, [1] presents a lightweight DBMS implementation in Coq proof assistant based on B+-trees. It consists of the relational algebra specification, its implementation in terms of proof trees and a set of verified query optimizations. The source code repository [2] contains Agda code of database schema specification and an implementation of relational algebra. The initial design of our model was inspired by this code. In [3] the strong typing for a relational database is provided in Haskell programming language. Haskell type system is less expressive than dependently-typed lambda calculus. Another example of relational DBMS specification in Coq proof assistant is available in [4]. The authors of [4] state that the specification and not the implementation as the primary objective of the research. Ur programming language [5] is worth mentioning as an example of richly typed programming language supporting the typing of SQL queries. None of the mentioned papers are con-

cerned with the representation and static checking of data consistency constraints which is the focus of the present paper.

4 Table schema and single-table constraints

The table schema in the present paper is defined analogously to [6]. The finite enumeration ‘AttrType’ specifies possible attribute type names (for example, integer number, floating-point number, string). The ‘el’ function matches each attribute type with the corresponding Agda type. Table schema is defined as a list of pairs (attribute name, attribute type). Type of table data ‘Row’ which depends on schema is defined as a tuple of elements of corresponding Agda types. For example, schema s = ((“Primary Key”, NATURAL), (“number”, STRING)) corresponds to row type Row s = N * String.

Constraints on separate table attributes are defined by means of ‘Constraint’ type which depends on the schema. Each element of ‘Constraint s’ type defines a single constraint on table rows. The sequence of rows in the table is defined as an inductive type with a constructor for an empty table and a constructor adding a row to the table together with the object proving that every constraint is satisfied for this row in combination with the previous rows. We can say that rows are chained in a table by proofs of the fact that all constraints hold. The “every constraint” is represented in the model in the terms of tuple with each element corresponding to the proof of the specific constraint conditions. A type representing the proof of constraint condition is constructed by means of ‘checkConstraint’ function. For the definition of the table structure and ‘checkConstraint’ function, the dependent types are required.

The simplest type of constraint is the constraint on the range of an attribute because the constraint checking involves a single attribute value. Such constraint ‘AttrValue’ in model is constructed with the following parameters: attribute index in schema ‘n’, the proof of attribute presence in schema ‘isAttr’ (static bounds checking), type of predicates over the type of the attribute ‘predicate’ and the decision procedure for the predicate ‘decision’. The constraint is satisfied if the proof of ‘predicate’ is supplied.

Constraints of uniqueness ‘Unique’ and primary key ‘PrimaryKey’ are defined just by an index of the corresponding attribute in the schema. These constraints are checked by iterating over each row in the table and verifying the uniqueness of the attribute value of the new row. Primary key constraint check includes also the requirement on a type of the attribute (natural number). In the future, the last constraint will be checked at the time of constraint declaration.

The set of single table constraints in the model can be extended to include the compound primary keys and arbitrary predicates on either a single row data or the contents of the whole table. The empty (“null”) values and the corresponding “not null” constraint is presently not supported, but the support may be added by means of technical changes. A more complicated task is the support of row update and row removal operations. For each change, the whole table must be reconstructed starting from the updated (or removed) row and each arbitrary whole-table predicate must be checked again. But as we mentioned before, the present paper is not concerned with

the efficient DBMS implementation. For the static constraint checking, we need to derive from the model the types of constraint checks on each predicate type.

5 Multiple-table constraints

As opposed to single-table constraints discussed previously, the foreign key constraint is defined on the set of tables. In the present section, we will call "table model" the pair of table schema and a list of the constraints over this schema. The set of database tables can be described as a list of pairs (table name, table model). This definition has some degree of similarity to a single table schema from the previous section: in both cases, it consists of the pair of the name (which can be used to declare constraints) and description of data. Following this similarity, the model elements described in the present section implement the constraint mechanism in the same way as in the previous section with a single exception: in this case data is not the tuple of the attribute values but the set of table rows. It is worth noting that such "two-dimensional" approach significantly increases the complexity of definitions from the technical standpoint. The main points of the approach stay unchanged: data is described as a log of insertions, each log entry is chained to the list of previous rows by means of constraint validity proofs. The insertion log entry for database consists of the name of the changing table, the contents of an inserted row, the previous state of the table and proof of successful constraint checks.

Foreign key constraints for the given set of tables 'd' consists of the five-tuple (t_1, a_1, t_2, a_2, pk_2), where 't_i' represent table names, 'a_i' correspond to attribute indices in the corresponding tables, and 'pk_2' is the proof of the property that 'a_2' is the primary key of the table 't_2'. The type of proofs of the foreign key constraint preservation after the row insertion into table 't_1' consists of the proof of the existence of row with the specified primary key in table 't_2'.

We are investigating the possibility of the addition of update and delete operations in case of multiple tables. The main obstacle in this task is the implementation of 'cut' operation which leaves only proofs relevant to the insertion operation for the specified table.

6 Statically checked Insert operation

On our way to useful in practice implementation of the proposed approach, we started by implementing the function 'StaticInsert' which inserts the row in the table with static constraint checks. This function presently supports only single-table constraints.

Our tests show that for constant data Agda type inference is able to provide the proof of constraints automatically. But for real code examples from ISTINA system, which works with tables with an unknown consistent state, such automated type inference is impossible. The future work includes the development of a method to obtain such proofs from database queries in application source code, as in the second example in "Related work" section.

7 Conclusion

The present paper includes the preliminary results of our research. An objective of this research is static (compile-time) checking of the fact that all domain constraints are respected in the source code. Such static checking will be implemented by means of source code translation to Agda proof assistant language in terms of the present model. The proposed model may, therefore, be used to check the code for different programming languages, including domain-specific languages for database interaction.

Acknowledgements
The first author is supported by RFBR Grant 16-07-01178a.

References

1. Toward a Verified Relational Database Management System / G. Malecha, G. Morrisett, A. Shinnar, R. Wisnesky. / Proceeding. POPL '10 Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages – 2010 – Pages 237-248 – DOI:10.1145/1706299.1706329
2. Agda Relation Algebra source code – 2014 – URL: https://github.com/sabauma/agda-relation-algebra (online; accessed: 2017-05-13)
3. Strong types for relational databases / A.Silva, J.Visser / Proceeding. Haskell '06 Proceedings of the 2006 ACM SIGPLAN workshop on Haskell – 2006 – Pages 25-36 – DOI:10.1145/1159842.1159846
4. A Coq Formalization of the Relational Data Model / V. Benzaken , E. Contejean, S Dumbrava / Programming Languages and Systems Volume 8410 of the series Lecture Notes in Computer Science – 2014 – Pages 189-208 – DOI:10.1007/978-3-642-54833-8_11
5. Adam Chlipala. Ur: Statically-Typed Metaprogramming with Type-Level Record Computation. Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10). June 2010.