Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 31 (2014) 369 – 378

Procedia Computer Science

2nd International Conference on Information Technology and Quantitative Management, ITQM 2014

Specification-based Approach for Denotational Semantic of Orthogonal Object/Relational DBMS

Amel Benabbou[a, c, *] , Safia Nait Bahloul [b,c]

[a] *Department of Computer Science, University of Abdelhamid Ibn Badis PB188, Mostaganem 27000, Algeria*
[b] *Department of Computer Science, University of Oran PB 1524, Oran 31000, Algeria*
[c] *LITIO: Laboratoire d'Informatique et des Technologies de l'Information, University of Oran , El-M'Nanouer, Oran, Algeria*

Abstract

The issue of the article is at the crossroads of databases modeling, software engineering and databases verification using formal methods. Development of databases software would be provided with a high-level specification suitable for formal reasoning about correctness properties. Formal specification techniques help discover problems in system requirements, inconsistencies and incompleteness can be resolved. Thus, we see a specified object/ relational database management system (ORDBMS) as a compelling challenge. Toward this goal, we propose a formal specification–based approach to describe a denotational semantic of an orthogonal object/relational model, a compiler for *SQL3* queries language and an implementation of execution engine of queries over imperative generic finite maps interface. This approach is of functional style based on inductive definitions and a high-order type theory realized within *Coq* proof assistant**.** Our work is a preamble step toward a verified ORDBMS.

Keywords: Formal specification, verification, denotational semantic, databases properties, proof assistant system, query processing.

1. Introduction

ORDBMSs involve the extension of relational databases systems to add object-oriented features. There are two major approaches for such integration, namely: stonebreakers[1] and Date & Darwen's[2] manifests.

The goal of software engineering in databases is to enable developers to construct DBMS that operate

[*] Corresponding author.
E-mail address: benabbou_amel@yahoo.fr.

reliably despite complexity. A way to reach this goal is by modelling them in a formal way[3]. Formal methods consist of a set of techniques that provide a mathematical framework in which it's possible to ensure the correctness of development[4]. They include formal specification to describe the system and its desired properties. Formal specification involves investing more effort in the early phases of software development. This reduces requirements errors as it forces a detailed analysis of the requirements. Formal specification is expressed in a language whose syntax and semantics are formally defined.

In general, our work fits toward the purpose of describing a denotational semantic for ORDBMS and to use it to verify that ORDBMS executes queries correctly according to such semantics. In this paper, we adopt a specification-based approach to describe formally a model, a compiler of queries language and execution engine of queries for ORDBMS using *Coq proof assistant*[5]. Our ORDBMS structures concepts are based on an object/relational model said orthogonal within the notation of Date & Darwen[2].

This article is organized as follows. Section 2 defines the problem. Section 3 is an overall presentation of our work. The definition of theoretical concepts of our object/relational model is given in section 4. In section 5 and 6, we present our formal specification of both model and SQL3 by proposing implementation of a compiler and execution engine for queries using Coq. Finally, a conclusion closes the paper in Section 7.

2. Problem statement and motivation

DBMSs are used to store data whose integrity and confidentiality must be strictly maintained. Application developer would be provided with a high-level specification for the behaviour of the data manager, suitable for formal reasoning about application-level security and correctness properties. Furthermore, the implementation of the data manager would be proven correct with respect to this specification to ensure that a bug cannot lead to accidental corruption or disclosure[7, 8]. ORDBMSs may be an appropriate choice for databases management that processes complex data and complex queries. Because of increase in complexity, the likelihood of subtle errors is much greater. For these reasons, and as first step, we see a specified ORDBMS as a challenge to the software engineering community that move beyond the domains of compilers semantics and theorem provers[9].

Definition of semantic increases understanding of systems by revealing errors and ambiguities. This makes it hard and tedious task. Therefore, it seems interesting to use specialized software to assist our work by means of Coq proof assistant. Coq is designed to develop mathematical proofs, especially to write formal specifications and implementations expressed in *Gallina* specification language.

We adopt the notation of Date & Darwen[2] which has the main advantage of being simple with orthogonal representation. That is to say, the formalism of Codd is maintained but enriched by a type system that integrates objects *independently* of relational concepts[10]. Different concepts are expressed in terms of the type system presented in[11].

3. Overall presentation

We give in first part the formal specification of some concepts of orthogonal object / relational model. The second part consists of formal specification of the queries language SQL3 which is the most appropriate on object-relational context. In the third part, we specified formally an execution engine for SQL3 queries.

In our ORDBMS, queries are written in an SQL3 subset form. It is supposed to guarantee the basic functionalities, namely: define headings and relation types, create relation variables, assign relations, load bodies, save and restore relations. Thus, we have implemented the principles steps given in Fig.1:

- Orthogonal object/relational model.
- A denotational specification for SQL3 complier including a lexer, parser, and queries optimizer.

- An execution engine implementation for SQL3 providing imperative operations using Coq's Ynot extension. The execution includes:

1. Parse SQL3 syntax into abstract syntax trees.
2. Transform the abstract syntax trees to algebraic expressions.
3. Optimizations on algebraic expressions.
4. Interpret queries as series of operations over finite map interface.

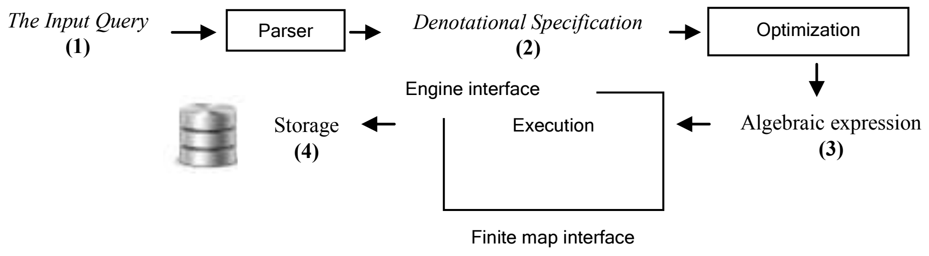

Fig.1. Specified components of our ORDBMS

4. Redefinition and formalization of basic concepts with Date & Darwen's notation

With Date & Darwen's notation[2], the semantic links between entities are modelled through *relation variables* and *relation types* concepts. We give our formal definitions adapted to Date and Darwen's notation as follows:

Definition 1 (Heading). A *Heading* {H} is a set of ordered pairs <A, T> such as:

a. A is the name of attribute in {H}.
b. T is the declared type of the attribute A.
c. Two pairs $<A_1, T_1>$ and $<A_2, T_2>$ are considered if $A_1 \neq A_2$

Definition 2 (Tuple). Given a collection of types T_i (i = 1, 2, ..., n, where $n \geq 0$), not necessarily all distinct, a *tuple* t -over those types- is a set of n ordered triplets of the form $<A_i,T_i,v_i>$, such as v_i is the value of the attribute A_i of type T_i.

Definition 3 (Body of relation). A *body* B_r of a relation *r* is a set of tuple t_i. However, there may be exist tuples t_j that conform to the heading {H} without that $t_j \in$ B.

Definition 4 (Relation). A *relation r* is defined by its heading $\{H_r\}$ and its body B_r. The Heading {H} represents the schema of the relation r. Relation of degree zero is valid.

Definition 5 (Relation variable). Given a heading {H}, the constructor type RELATION {H} generates the relation type *tr* with the heading {H} to define a *relation variable* "*relvar*" of type *tr*. The Defined relation variable has the same heading {H} as the relation type *tr*. For example:

```
VAR S RELATION {S # S#, SNAME NAME, STATUS INTEGER, CITY CHAR};
```

5. Formal specification of object/relational model

In this section, we describe a formal specification by choosing an appropriate encoding for an orthogonal object/relational model. Using an adequate efficient environment for formal reasoning, we describe in informal way the concepts across Date & Darwen's notation and how specifying them within Coq.

In our object/relational model, database is modelled using relations by instantiation of relation variables. A relation is represented by some heading and body. As Coq is a fully typed environment, we consider that the heading is *list* of types, known as the *schema* of the relation. The body is a set of *tuples* indexed by a set of attribute names. For simplifying, we use the position of element as the attributes name. We represent relations as finites sets of types. Finite sets are a common abstraction and Coq provides them as a standard library. Within Coq, we provide formal specification of the basic concepts as follows:

5.1. The Heading

We see the heading as the schema of a relation, defined as a list of types and denoted *tnameHeading* presented within Coq as the inductive definition:

```
Inductive tnameHeading: Set: =
|Integer: tnameHeading
|Boolean: tnameHeading
|Char:    tnameHeading
|TUPLE:   tnameHeading
|RELATION:tnameHeading
   ...
|Option:  tnameHeading -> tnameHeading.
```

Option constructor is used to denote a particular type that contains no values (null values" ǀ ") presented in [12] and treated within a type system given in[11]. Our type names can be mapped to Coq types by the functional denotation *tnameDenote*.

```
Fixpoint tnameDenote (t: tnameHeading): Set :=
match t with
     | Integer => Z
     | Boolean => bool
     | Char => string   ...
     | Option t' => option (tnameDenote t')
end.
```

In Coq, generally, *option* types are used instead of *not.found exceptions*. Our idea of choosing *option* types is due to try extending the partial function *Option t* by the special value ǀ such that *Option t'* : *tnameHeading* → *tnameHeading* ∪ { ǀ } is a total function. *Option t'* returns the current type name of an attribute or *None* if no type exist. Z, bool, char are the corresponding Coq types. Thus, we give the formal specification of schema of relation as follows:

```
Parameter tnameHeading:  Set.
Parameter tnameDenote: tnameHeading -> Set.
Definition Schema: Set:= list tnameHeading.
```

5.2. The Body

A tuple is a heterogeneously-typed list of pairs (*v,t*) terminated by *unit* type to mark up the end of the list by *nil*. The formal specification of a tuple on schema S is defined within Coq by the recursive function Tuple and given as follows:

```
Fixpoint Tuple (S: Schema): Set:=
    match S with
         | nil => unit
         | v :: t => tnameDenote v * Tuple t
end.
```

For example, the following tuple:

```
Definition   aTuple: Tuple MySchema :=
          (100,"James", (true, tt))).
```

May be defined on the schema:

```
Definition MySchema:  Schema:=
      Z:: char :: Bool :: nil.
```

We implement a few operations on tuples. For example, the function that performs the product to fuse tuples:

```
FJoinTuples (I J: Schema)(t: Tuple I)(t’: Tuple J) :
Tuple (I ++ J).
```

We also use the richness of Coq to help simplify reasoning about error cases. For instance, to project out the type name of a particular attribute A (represented by the position *n*) from a schema *I*, we need to provide a proof *pf* that *n* is less than the length of *I*:

```
attType (I:Schema) (n:nat) (pf: n < length I) : tname.
```

5.3. Relations

We see relations as a finite set of types (the body). Then, the choice that we consider is how to represent finite sets in Coq. Finite set are given in coq as "*FSets*" library. But unfortunately, we cannot use it directly because it is coded as compile-time functor parameterized by a fixed type, which is restrictive for our ORDBMS where the type parameter must be determined at run-time, after assigning relations to relation variables. For this, we modify the *FSet* library to be first-class using Coq's type class mechanism[13]. A type class is a set of functions specified for a parametric type but defined only for some types. It allows sharing notations by overloading operations and specifying with abstract structures. We define the class of types *FSetInterface* as a common name for different structures that implements finite sets. An instance of *FSetInterface* for an ordered type *E* contains the type *FSet* of elements *elt* and all the operations that these sets support. We give a part of such formal specification as follows:

```
Class FSetInterface (elt: Set) (E: OrderedType elt)
: Type:=
{Fset : Set; (* the container type of finite sets *)
       (* operations *)
empty: Fset;
union: Fset -> Fset -> Fset;
inter: Fset -> Fset -> Fset;
is_empty : Fset -> bool;
add : elt -> Fset -> Fset;
 ...
       (* predicates *)
In : elt -> Fset -> Prop;
Definition Equal `{Set elt} s s' => forall a
: elt,
In a s <-> In a s'; (* In is the membership function*)
...
       (* axioms *)
union_1: forall s s' x, In x (union s s')-> In x s \/ In x s';
(* refer to the In function*)
union_2 : forall s s' x, In x s -> In x (union s s');
...
}
```

Thus, formal specification of a relation in our model is defined over finite sets of schema typed tuples. Building relations requires defining a total ordering over tuples and interacting with the type class mechanism, it is given as follows:

```
Definition Relation (I: Schema):=
FSetInterface (Tuple I).
```

6. Formal specification of a compiler for SQL3 queries

The task of query compiler is to generate a query evaluation plan that can be executed. In what follows, we provide a first insight into the workings of a SQL3 compiler and query processor.

6.1. SQL3 queries compilation process

The SQL3 compiler performs several steps to produce an access plan for a given query. These steps are shown as follows:

6.1.1. The lexer

Our SQL3 processor does not see the query as a whole but consumes user input as a stream of characters. Some characters do not contribute to the meaning of the query but rather define “word” boundaries, like whitespaces. Our SQL3 scanner drops whitespace, and maps sequences of characters to tokens.

6.1.2. The parser

We are interested in syntax analysis to determine that the structure of a query is correct and to convert it into abstract syntax trees for further processing. In *SQL3*, we reference relation by a name which is represented as strings, so it is explicitly typed. We can define a query within Coq as inductive type with relational operations as type constructors (union, projection, etc). This type is given as follows:

```
Inductive Query : Schema -> Set :=
| relvarExp : forall I, string -> Query I
| unionExp : forall I,
Query I -> Query I -> Query I
| diffExp : forall I,
Query I -> Query I -> Query I
| restrictExp : forall I,
whereExp I -> Query I -> Query I
| projExp : forall I (l:list nat)(pf: bounded I n),
Query I -> Query (attTypes l pf)
| joinExp : forall (I J: Schema),
Query I -> Query J -> Query (I++J).
```

We use Ynot packrat PEG parser[14] to parse user input. This parser is implemented as a verified compiler. For more efficient parsing, the packrat algorithm uses sophisticated strategies implemented using hash tables. We give next the specification of *whereExp* with boolean combinations of comparisons between attributes and values:

.

```
Inductive atomicExp (I: Schema): tname -> Set:=
| const: forall t (c:tnameDenote t), atomicExp I t
| att : forall n (pf: n < length I),
atomicExp I (attType I n pf).

Inductive compareExp I: Set :=
| compareEq : forall t, atomicExp I t ->
atomicExp I t -> compareExp I
| compareLt ...

Inductive whereExp I :=
| compExp: compareExp I -> whereExp I
| andExp : whereExp I -> whereExp I -> whereExp I
| orExp ...
```

6.1.3. *SQL3 semantics*

The main purpose of semantic analysis is to associate a type to every expression. In our ORDBMS, the semantic of a SQL3 queries is defined in terms of relational algebra. It consists to know the currents relations assigned to relation variables (*relvar*). These relations are of a heading equal to that of *relvar* and a body consisting of tuples loaded par user. *Relvar* names are represented as strings and are explicitly typed. We can associate *relvar* and relations by the mechanism of *context* and *environment*, respectively, as follows:

```
Definition Ctx := list (string * Schema).

Fixpoint Env (G: Ctx): Set :=
match G with
| nil => unit
| (_, J) :: b => Relation J * (Env b)
end.
```

When an environment has current relation that corresponds to relation variable in a query, the denotational semantics of a query is defined then recursively by applying the relational operations:

```
Fixpoint denote (I: Schema) (q: Query I)
(G: Ctx) (E: Env G): Relation I:=
match q with
| varExp J v => lookup E I v
| unionExp J a b =>
union (denote a) (denote b)
| diffExp J a b =>
diff (denote a) (denote b)
| restricExp J r f =>
 (whereDenote f) (denote r)
| projExp J l pf e => proj l pf (denote e)
| joinExp I' J' a b =>
join (denote a) (denote b)
end.
```

6.1.4. *Query optimization*

It consists of a semantic-preserving transformation that reduces the relation size and execution time. Our SQL3 query optimizer takes the parsed representation of SQL3 query as input and generates an efficient execution plan. For example, an optimization that reduces the number of joins and pushes restriction toward the end. We implement few queries optimizations with more simple syntax of restriction, projection and join operations, adopting the notation of Date and Darwen's query language[2]:

- **Restriction fusion**. The query (r_1 Where P_1) Where P_2 can be transformed to a single restriction: r_1 Where (P_1 and P_2).
- **Projection fusion**. The query r_1 {All But L_1} {All But L_2} can be transformed to a single projection r1{All But L_1 ,L_2 }.
- **Join equivalence**. The query r_1 JOIN (r_2 JOIN r_3) is transformed to JOIN {r_1, r_2,r_3 }.

Our study of queries optimization takes into account semantics optimization. We aim to find equivalent query rewritings with more efficient evaluation plans[15]. For example, we consider the following "project – join" equivalences for relations r_1 and r_2 with attributes: x_1, x_2, x_m,…y_1, y_2,…,y_n and y_1,y_2 ,y_n… z_1, z_2,...,z_p respectively.

- r_1, $r_2 \neq$ empty and $p = 0 \rightarrow$ (r_1 JOIN r_2) All But z} ~ r_1 Semijoin r_2
- r_2 = empty $\rightarrow$ (r_1 JOIN r_2) {All But z} = r_1

Can be used for such optimization in which rewrites depends on relations emptiness constraints that must be verified. The emptiness information is provided to semantic optimizer with the available emptiness proof. We use Coq's record structure called *RelationInfo* to implement that semantic information:

```
Record RelationInfo := relInfo {isEmpty: bool}
```

Furthermore, each relation has the meta data *MdInfo* which consider relation name (string), schema and also relation information:

```
Definition MdInfo : string -> Schema -> RelationInfo
Definition sem_rewrite: Set:= MbInfo -> (forall I, Query I
-> Query I).
```

In our example (r_1 JOIN r_2) {All But Z}, the optimization identifies the adequate evaluation expressions by using *MdInfo* to decide which rewrite to apply. Finally, we must ensure that semantic optimization is used correctly, i.e., make sure that the associated *RelationInfo* information is accurate. It is necessary to prove that the semantic optimization is meaning preserving. Thus, we define an accurate state of our database as follows:

```
Definition StaAccu (m: MdInfo)
(G: Ctx) (E: Env G): Prop :=
forall s I, getRelation s I E = empty <->
isEmpty (m s I).
```

The advantage of our semantic optimization is that the optimizer can infer the information about intermediate data from semantic knowledge prepared prior to query execution time. It makes less risky because we are required to prove semantic preserving in a formal way using Coq.

6.2. Execution engine for SQL3 queries

Our query execution engine implements a set of operators that take the optimized queries as input. We execute SQL3 queries using a sequence of operations over imperative *finite maps* (associative array) composed of a collection of pairs (*key*, *value*) with the usual operations (*insert*, *lookup*, etc). We give the implementation of finite maps in terms of a generic interface, i.e, we realize this implementation such that it can be done by any data structure (like B+ trees or table hash). We describe in what follows the finite map interface and how SQL3 queries operations are interpreted within.

6.2.1. Specification of the Finite Map Interface

We describe the finite map interface with *functional* and *imperative* specifications. The functional one is to represent finite maps as associated lists key-value, given as follows:

```
Parameters key value: Set.
Definition AssociatedList:= list (key * value).
```

The state of operations such as *lookup* is described over functional Abstract Data type (ADT) given in Coq by a functional definition as follows:

```
Fixpoint specLookup (k : key) (m : AssociatedList)
: option value :=
match m with
| nil => None
| (k',v) :: b => if k' = k then Some v
else
specLookup k b
end.
```

The imperative specification is to represent operations over imperative ADT which we call *Process* that represents a handle on the imperative state of the finite map. We have realized our imperative specification using Ynot library (an axiomatic extension to Coq that permits writing and reasoning about imperative code).

This specification is of *Hoare Type Theory* style[16] allowing the use of commands indexed by pre- and post-conditions to describe operations. A portion of the ADT process specification is given as follows:

```
(* The abstract type of imperative finite maps *)
Parameter Process: Set.

(* The predicate rep p m holds in a heap when the finite map p   represents the association list m *)
Parameter rep: process -> AssociatedList key value -> heap -> Prop.

Parameter insert: forall (p: process) (k: key) (v: value) (m:
AssociatedList),
Cmd (rep p m) (fun res: option value => rep p (specInsert v m) * [res
= specLookup k m]).

Parameter iterate: forall (T: Type) (p: process) (I: T ->
AssociatedList -> heap -> Prop) (tm: AssociatedList) (acc : T)
fn: forall (k: key) (v: value) (acc: T) lm, Cmd (I acc lm) fun a: T
=> I a (lm ++ (k, v):: nil)))...
```

For example, the *insert* operation takes a process p, key k, and value v such that p represents association list m on input, and it ensures that p represents the result of inserting (k; v) into m on output. The command also returns the old value that was assigned to k in the input state; this makes it easy to undo the insertion.

6.2.2. Interpretation of SQL3 operations

As our SQL3 engine executes queries using finite map interface. We execute queries by interpreting them as a sequence of finite map over the ADT *Process* operations as *iterate* and *insert*:

- Restriction. To restrict over $r_{1,}$ *iterate* through r_1 and *insert* tuples that verify the predicate into the new relation.
- Projection. To project from r_1, *iterate* through r_1 and *insert* the projected tuples into a new relation.
- Union. To union $r_1$and r_2, *iterate* through r_1 and *insert* each (k; v) $\in r_1$ into r_2.
- Join. To join $r_1$and r_2, iterate through each and, for each t $\in r_1$, *iterate* through each t'$\in r_2$, *insert* a fused tuple t + t' into the new relation.
- Difference. To minus $r_1$from r_2, *iterate* through r_1 and *insert* each tuple t $\in r_1$ into a new relation when t $\notin r_2$.

Conclusion

In this paper, we have given formal specification of some key concepts using Coq[5]. Implementation of object/relational concepts adopting the notation of Date & Darwen is in full experimentation within our works actually. Thus, verification of such system appears necessary to prove its reliability in terms of correctness of development in complex environments. Using software to assist formal specification and verification has a great impact in whole process of development. Our experience shows that though many challenges remain, but we confirm that building fully-verified systems software in Coq is within reach.

References

1. Michael, S.: Object-Relational DBMSs: The Next Great Wave. Morgan Kaufmann 1996, isbn 1-55860-397-2.996.
2. Date, C.J., Darwen, H.: Databases, Types, and Relational Model: The Third Manifesto", (3 rd edition); Addison- Wesley, 2007
3. Vangalur, S., Alagar, V., Lakshmanan, V., Sadri, F.: Formal Methods in Databases and Software Engineering. In Proceedings of the Workshop on Formal Methods in Databases and Software Engineering, Montreal, Canada,1992
4. Sonali, U., David, B., Dinolt, G., Levin, E.: Evaluation of Program Specification and Verification Tools for High Assurance Development. Naval Postgraduate School (U.S.) 2003-09-00 Information Systems Security Studies and Research (CISR) Papers.2003
5. The Coq Proof Assistant: http://coq.inria.fr/
6. Nanevski, G., Morrisett, A., Shinnar, P., Govereau, L.: Ynot: Dependent types for imperative programs. In Proc. ICFP, 2008
7. Frey, R., Radhakrishnan, H.: A Formal Specification and Verification Framework for Time Warp-Based Parallel Simulation. IEEE Transaction on Software Engineering. Vol. 28, No. 1, January 2002.
8. Bayley, A., Zhu, H.: Formal specification of the variants and behavioural features of design patterns. Journal of Systems and Software 83(2): 209-221, 2010
9. Bertot , Y., Castéran , P., Huet, G., Paulin-Mohring, C.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions" (Texts in Theoretical Computer Science). Springer; 1 edition, 2004.
10. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM 13, no. 6, 377{387. 1970
11. Benabbou, A., Nait Bahloul, S., Amghar, Y., Rahmouni, K.: An Algorithmic Structuration of a Type System for an Orthogonal Object/Relational Model. CoRR abs/1007.3275: 2010
12. Benabbou, A., Nait Bahloul, S., Amghar, Y., Rahmouni, K.: Semantic expression of incomplete information in the object / relational model, 1st SIGSPATIAL ACM GIS 2009 International Workshop on Querying and Mining Uncertain Spatio-Temporal Data November 3, 2009, Seattle, WA, USA.2009
13. Sozeau, M., Oury, N.: First-Class Type Classes. In Otmane Ait Mohamed, C.M., Tahar, S., eds.: Theorem Proving in Higher Order Logics, 21th International Conference. Volume 5170 of Lecture Notes in Computer Science. Springer 278-293, August 18-21, 2008.
14. The Packrat Parsing and Parsing Expression Grammars Page. http://bford.info/packrat/
15. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL Query Optimization. In ICDT, 2010.
16. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in hoare type theory. In Proc. ICFP, 2006.