Models & Optimisation and Mathematical Analysis Journal Volume 01 Issue 02 (2012)

Formal Specification by Coq of Date and Darwen's Object/relational Model

Amel Benabbou
LITIO Laboratory – University of Oran
BP 1524, El-M'Naouer, 31000
Oran, Algeria
benabbou_amel@yahoo.fr

Safia Nait Bahloul
LITIO Laboratory – University of Oran
BP 1524, El-M'Naouer, 31000
Oran, Algeria
nait-bahloul.safia@univ-oran.dz

***Abstract*— Development of databases software systems would be provided with a high-level specification, suitable for formal reasoning about application-level security and correctness properties. Formal specification is a key element of formal methods. It can greatly increase comprehension of a system by revealing inconsistencies, ambiguities, and incompleteness that might occur. Thus, implementation would be proven correct with respect to this specification to ensure that a bug cannot lead to non-conformance of properties or accidental corruption. It is for these reasons that we see verified DBMSs as a compelling challenge to the development of software. As a step toward this goal, we establish in this paper formalization through a formal specification of some basis concepts in object/relational model proposed by Date and Darwen, using the *Coq* proof assistant system. We give the challenges acquired from our experience using that proof tool. Our work is a preamble step toward a fully-verified object/relational DBMS.**

***Index Terms*—Formal specification, proof assistant system, Coq, verification, object/relational model.**

introduction

Software databases systems inevitably grow in scale and functionality. Because of this increase in complexity, the likelihood of subtle errors is much greater [1]. A major goal of software engineering is to enable developers to construct systems that operate reliably despite this complexity. One way of achieving this goal is by modeling them in a formal way [2, 3]. Formal modeling plays a key role in building of high assurance and trusted software databases systems. This modeling enables formal reasoning about the system design that can be used to prove that the system has certain properties [4, 5]. A distinguishing feature of high assurance systems is that they are modeled mathematically using formal methods [6, 7].

Formal methods consist of a set of techniques and tools based on mathematical modeling and formal logic that are used for *specification*, development, proof and verification requirements and designs in software systems, such as in databases management systems (DBMSs). Particularly, they provide a mathematical framework in which it is possible to ensure the correctness of development and assurance in the validity of the results [8]. By defining languages with a clear semantics, and making explicit how to reason on these later.

Specification is a key element of deductive formal methods; it's the process of describing a system and its desired properties [9]. Formal specification uses a language with mathematically defined syntax and semantics.

One current trend is to integrate different specification languages, each able to handle a different aspect of a system. Another is to handle non-behavioral aspects of a system such as its performance, security policies, architectural design, and theoretical concepts, integrity constraints, functional dependencies in Databases [10, 11, 12]. Formal specification can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected. Therefore, it seems very interesting to use specialized software to assist in the specification by means of *Proof Assistant system* **[**13, 14, 15]

The goal of this paper is to give a formalization of some key concepts of the orthogonal object/relational model of Date and Darwen proposed in [16], the different concepts are expressed in terms of the type system which we have presented in [17]. This type system has a pseudo-algorithmic and grammatical description of all types in such model, namely: scalar, tuple, and relation types. Our algebraic grammar describes in detail the complete specification of an inheritance model: simple and multiple. An extension of this type system is performed by a special type representing null values [18]. Such an extension is prompted by a position that favors the existence of null values in object / relational model and proposes a semantic expression. We describe, specially, a formal specification for these concepts using Coq Proof Assistant [19].

Coq Proof Assistant is designed to develop mathematical proofs, and especially to write formal specifications, implementations and to verify using type-checking algorithm that the later are correct with respect to their specification. Thus, it allows interactively constructing formal proofs and supports specification of static data, functions and definitions which can be developed using the basic specification language Gallina in Coq. Using Curry-Howard isomorphism [20, 21], programs, properties and proofs are formalized in the same language called Calculus of Inductive Constructions [22], that is a λ -calculus with a rich type system [23, 24]. Conveniently, the reasoning is at the same platform with the specification. All above strong points of Coq make us to use it to finish our specification work.

This article is organized as follows. Section 2 is a definition of theoretical concepts of Date and Darwen's data model. In section 3 we present our formal specification for most important relational concepts in that model using the proof system Coq. In section 4 we discuss the challenges we faced using that proof environment of development in building such specification. Finally, a conclusion closes the paper.

Definition of basic concepts in date and darwen's data model

Date and Darwen have proposed a theoretical basis for integration of a few object concepts in the relational context [16]. In that model, it's not necessary to restructure the model to achieve the object concepts. It is enough to expand domains to new abstract data types, and allow inheritance and sub-typing in order to take advantage of the object-orientation features.

Date and Darwen have also proposed a query language *Tutorial D* and relational algebra *A*. The semantic links between these entities are modeled through the concept of relation variables (*relvar*). The DBMS *Rel* implements a significant portion of Date and Darwen's *Tutorial D* query language.

Thus, Date and Darwen have given formal definitions adapted to their vision of future databases. We present such definitions 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.

Given a heading {H}, a relation variable *relvar* must be of type RELATION {H} [16]. The instantiation of a relation variable *relvar* is done explicitly during the operation of defining *relvar*, or an empty relation if no explicit value is specified.

Formal specification of basic concepts by Coq assistant proof

In this section, we describe a formal specification by choosing an appropriate encoding of the object/relational model and using an adequate efficient environment for formal reasoning and specification, namely Coq assistant proof system.

Formalization of basic concepts is given for many reasons:

- Basic concepts and algebra make foundation constructs of the model, indeed; their specification establishes formal semantics that conspire to take a formal and rigorous comprehension of these concepts.
- Specification of a concept is strongly linked to another concept such as with relation and tuple.
- Specification of queries is based on the specification of the basic concepts (a query concerns necessarily a relation).
- Passing to verification tasks is necessarily preceded by a detailed specification of every concept.

Relational algebra has a standard definition in terms of set theory [25]; therefore, we consider that our work of specification deals with realizing both sets and relational algebra (see section D) in coq. We describe briefly in an informal way the key relational concepts in that model and how specifying them within Coq.

Object/relational model of Date and Darwen is modeled using relations. In such model, a relation is represented by some *heading* and a *body* which is simply a finite set of tuples over a *set* of couples (attribute, type). To simplify our work and accordance the typing environment of Coq, we consider that an attribute in a heading is represented by its type. So, the heading is described as a *list* of types. The list of types that describes the attribute is then known as the *schema* for the relation. *Tuples* in a relation are indexed by a set of attribute names. Again, for reason of simplification, we use the position of an element as the attribute name.

There are many ways to represent relations in Coq [19]. For example, in [26] it is suggested that schemas should be represented as functions from a finite set of attribute names to type names, but in practice, we found that a concrete encoding using a list of type names yields a more workable representation. Another choice was how to represent relations as finite sets. Finite sets are a common abstraction and Coq conveniently provides them as a standard library.

We provide now formal specifications of these concepts using Coq assistant proof system:

A. *Schema of relation (the heading)*

We define the schema for a relation as a list of type names which denoted tnameHeading. Each type in the heading are is generated recursively from a type system defined in [17] and we might define tnameHeading as the inductive definition:

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

We must indicate here that “Set” refers to Coq’s predefined *sort* or *type universe* [19], not sets in the sense of relations (in Gallina language, the term tnameHeading is called *specification*). Type TUPLE and RELATION will be given in next sections. *Option* constructor is used to denote a particular type that contains no values (null values ”⊥ ”) which we have presented in [17] and treated within the type system given in [18] through our in-depth study of Date and Darwen’s object/ relational model [16].

Type names can be mapped to Coq types by a denotation function tnameDenote. The definition for type names below and the denotation function are parameters to the system so that users can easily add new constructors to the set of schema types. We define this denotation function as:

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

The function tnameDenote is defined as recursive (via Fixpoint coq’s key world). In this definition, tnameDenote take t as argument of type tnameHeading on which recursion is organized, and return some type represented as universe “*Set*” since tnameDenote a general function for defining types. defined tnameDenote recursive functions contain a *filter* (with coq’s *match* key word) on the argument t, so that it is led to give in fist the value of the function when the argument is nat, boolean, and so on, then the value of the function when the argument is of the form "*option* t’ " with the possibility of using the value of the same function in t '.

In Coq, generally, option types are used instead of *not.found exceptions*. Our idea of choosing use of option types is due to try extending the tnameHeading of the partial function Option t: tnameHeading → tnameHeading, by the special value⊥ such that Option t’ : tnameHeading → tnameHeading ∪ {⊥ } is a total function. Option t’ return the the current type name for an attribute or *None* if no such type exist. Z, bool, etc are the corresponding Coq types. The values that make up tuples are inhabitants of the denoted Coq types.

Thus, we give the overall formal specification of a schema of relation (represented as a list of type names) as follows:

```
Parameter tnameHeading  :  Set.
Parameter tnameDenote  :  tnameHeading ->
Set.

Definition Schema :  Set  := list tnameHeading.
```

Date and Darwen’s 8th RM[1] prescription entitled EQUALITY has affirmed that Tutorial D shall support the equality comparison operator for every type T. We need then to be able to compare schemas for equality. Equality between arbitrary Coq types is undecidable, so we require decidable equality on type names as another parameter to the system, and is expressed with the following specification:

```
Parameter  tnameHeading_dec_eq :
      forall (x y: tname), {x=y} + {x< >y}.
```

Additionally, formalizing in dependent type theory often requires representing sets as setoids, i.e. types with an explicit equality relation. Thus, we require that for any type name, the Coq type T that it denotes satisfies the following properties (which we need in our future work for queries optimization verification):

1. T must be a decidable setoid .i.e., equipped with a decidable equivalence relation.
2. T must be a decidable total order .i.e., equipped with a decidable total ordering compatible with the setoid.

Like decidable equality for type names, these properties on denotations are given as parameters to the system: Property (1) allows for equivalence relations on attributes types that are weaker than syntactic Leibniz equality. Property (2) is required because of the way we build sets of tuples,

[1] RM: Relational Model

B. Tuples (the body)

Each attribute in a relation is of some type, a tuple associates a value of the appropriate type to each attribute. That is, a tuple is a heterogeneously-typed list. The type of a tuple is given by a recursive, type-level function Tuple parameterized by some Schema S (defined above). In Coq, we specify a tuple as a list of pairs (value, type) represented here as (v,t), constructed by Cartesian product of latter pairs and terminated by Coq's *unit* type in order to mark up the end by *nil*. *Unit* is the Coq's singleton data type that contains a sole inhabitant written *tt*, and predefined as:

```
Inductive unit : Set :=
    tt : unit.
```

The Coq system provides a notation for lists, so that "cons v t" is noted "v :: t" (where:: is list cons). Formal specification of tuple type is given as follows:

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

Tuples of the body of some relation are essentially iterated pairs of values terminated by a unit. For example:

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

A tuple on such schema may be:

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

We need several tuple manipulation functions in order to express the relational operations. For example, to perform product of tuples, we define the function FProdTuples that realize the operation of fuse tuples. The type of this function ensures that the schema of the resulting fused tuple is the concatenation of the input schemas:

We also use the richness of Coq's type system 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.
```

The operation of project a single attribute from a tuple uses attType in its type:

```
projTupleAtt (I: Schema) (n: nat) (pf: n < length I)
  (t: Tuple I) : tnameDenote (attType  I   n  pf).
```

C. Relations

We can see relations as a finite set of types. Coq provides the "*FSets*" library composed of several packages. The library is coded as a standard ML-style functor i.e., as first order parametric *module*, which requires the static determination of the corresponding module's signature[2].

Then, the choice that we consider is how to represent finite sets in Coq. We assume that we could not use the "*FSets*" library directly. Because in *Rel* DBMS, the relation type must be computed from a schema at run-time, not before; obviously, it does not know table schemas until the user loads data at run-time (since modules signature has been already defined). For this, rather than try to encode such behavior using just *modules*, we modified the FSet library to be first-class using Coq's type class mechanism [27].

Type classes are a recent addition to Coq presented as a solution to allow user overloading notations, operations and specifying with abstract structures by quantification on contexts[3] across a class of types. They behave similarly to their Haskell counterparts which have been introduced to make ad-hoc polymorphism less ad hoc [28]. In informal semantic similarities description, we can say that Coq's type class plays the role of relation variable *relvar* presented in Section II. Classes instantiation is similar to the process of assigning some relations values to a given *relvar* having the same heading as also the same applied operators.
We establish our work of specification on the basis of the Library *Containers.SetInterface* that defines the interface of typeclass-based finite sets. We have a class of types *FSetInterface* that is parameterized by a type elt of elements and a total ordering *E* over *elt* that can be used as specifications of finite sets. Here "*Prop*" indicates Coq's *proposition* sort:

```
FProdTuples (I J: Schema)(x: Tuple I)(y: Tuple J)
:
Tuple  (I ++ J).
```

```
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;
...etc
 (* the 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*)
Definition Empty `{Set elt} s :=
  forall a : elt, ~ In a s.
```

We define the class *FSetInterface* of structures that implement finite sets. An instance of *FSetInterface* for an ordered type E contains the type *FSet* of elements *elt*. It also contains all the operations that these sets support: insertion, membership, etc. The specifications of these operations are in a different class. In addition, we specify a set of axioms that allow us to reason about the operations.

Thus, formal specification of a relation in Date and Darwen's 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, and it is given as follows:

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

We must also indicate that this specification is realizable otherwise, which we can do by providing a simple implementation using lists. In this case, we require the element types to be ordered. An alternative implementation of a finite set would be as a sorted list with a proof that the list contains no duplicates.

In general, we have found that the richness of Coq, including support for ML-style modules, dependent types and type classes, coupled with abstraction and equality issues yields a set of tradeoffs that are difficult to evaluate.

D. Relational Algebra A

Date and Darwen have described a new relational algebra *A* [16]. The algebra *A* differs slightly from Codd's original algebra in some aspects but it is identical in a great part.

As defined by Date and Darwen, a database is modeled using relations. We can represent a relation as a finite set of tuples over a list of types [17]. New relations are constructed using a basis of operations: *Selection, Projection, Union, Permutation, Difference, and Cartesian product.*

This basis is relationally complete, and equal in expressive power to other relational formalisms, like relational calculus. We define the relational operations over relations. Then, we consider that formal specification of these relational operations will be given in terms of coq's predefined functions. Though, we modify the definitions in such way that it will be adapted with the light changes introduced in Date and Darwen's definitions of Algebra A.

Union, *difference*, and *selection* are implemented in terms of the *FSetInterface* union, difference, and filter functions, respectively. In Tutorial D, union dyadic r_1 UNION r_2 (where r_1, r_2 have the same headings) is semantically equivalent to the algebra A expression r_1<OR> r_2.

Projection and *product* are defined using the generic fold function provided by the *FSetInterface*. *Selection* allows any Coq predicate that respects the setoid equality of the schema to be used.

Projection is implemented by iterating through a set, projecting out each tuple individually. *Cartesian product* is slightly more complicated, requiring two iterations. To compute the product of two relations r_1 and r_2, for each $x \in r_1$ we compute the set {FProdTuples x y | $y \in r_2$} and then union the results.

We have indicated that our main mission will concern the verification task, that is showing that the *Rel* DBMS executes correctly queries with respect to a denotational semantics of Tutorial D and relations. Therefore, we need some lemmas to support basic reasoning.

To prove the accuracy of our specified relational Algebra, we have shown that several standard equivalences are derivable from our definitions. Some equivalences are universally valid; for example, the commutativity of *selection*:

Select P1 (select P2 R) = select P2 (select P1 R)

Other equivalences only apply in the presence of constraints on relations. For example, let r_1 and r_2 be relations over schemas I and J, respectively, and let l indicate the attributes 0…|l|- 1. We have the conditional equivalences:

```
r₂ <> empty -> proj l (prod r₁ r₂) = r₁
r₂ = empty -> proj l (prod r₁ r₂) = empty
```

Proving this statement requires reasoning about how projection of l can be effectuated through the nested iteration that defines the product. it may be possible to adapt an automated theorem prover for relations (e.g., [29]) to Coq to reduce the proof complexity.

Challenges

In what follows we highlight some challenges from our work of specification.

Use dependent types in our definitions was a source of difficulties. We found dependency useful to express schemas for operations and to rule out various error cases that would arise. Newer languages such as Epigram [30] provide better support for programming with dependent types. For Coq, several works ongoing to adapt many of these ideas, so we are hopeful that these difficulties will diminish.

Another challenge is that the Coq modules are useful for controlling name spaces, but their second-class nature makes it difficult to use them effectively for abstraction. Rather, we found core language mechanisms, such as dependent records and type classes, to be more useful than modules. Consequently, we avoided sophisticated use of the module system when possible.

A final challenge is the formalization of algebra A. The inspiration for our work, however, is the formalization of the relational algebra in Agda found in [26]. Like that work, we use axiomatic finite sets; however, we opted for a more concrete tuple representation and a different, but equivalent, choice of base operations.

Conclusion

In this paper, we have given a formal specification of some key concept of that model using one of the most famous proof assistant system namely Coq proof assistant [19].

We declare that we have settled for merely a few basic concepts but our study of formalization and formal specification constitute a preamble step to a complete demarche toward a verified object / relational data model and subsequently toward verified data bases management system.

The implementation of the concepts of Date and Darwen in a DBMS is in full experimentation around the world, for instance the DBMS *Rel* that implements a significant portion of Date and Darwen's Tutorial D query language. Thus, the verification of this system appears necessary to prove its power and reliability in terms of correctness of development and in terms of security requirements during its running on complex environment and applications. What's persists in our future work concerns especially the verification tasks, that is to say verifying that queries in *Rel* are executed correctly according to their specification.

Writing a formal specification is already an improvement compared to standard approaches. Indeed, by using Coq proof assistant system, many ambiguities are resolved. Furthermore, Coq provide ways to check consistency of the specification allowing large complicated proofs. Thus, using software to assist formal specification and verification has been of great impact in whole process of development.

References

[1] S. Vangalur, Alagar, V. S. Lakshmanan, F. Sadri (Eds.)," Formal Methods in Databases and Software Engineering", Proceedings of the Workshop on Formal Methods in Databases and Software Engineering, Montreal, Canada, 15-16 May 1992.

[2] É. Jaeger, T. Hardin,"A Few Remarks About Formal Development of Secure Systems", CoRR abs/0902.3861: 2009.

[3] A. van Lamsweerde," Building Formal Models for Software Requirements", APSEC 2000: 134- 7th Asia-Pacific Software Engineering Conference (APSEC 2000), 5-8 December 2000, Singapore. IEEE Computer Society 2000.

[4] D. Bell, D, "Foundations and Model". M74- 244, The MITRE Corp., Bedford MA,E. and LaPadula, L.J. Secure Computer Systems: Mathematical. May 1973

[5] W. Cheng, X. Zhang, J. Liu, "A Secure Policy Model for Secure Database System Based on Extended Object Hierarchy", Journal of Software, vol.14, No.5, 2003.

[6] U. Sonali; Bibighaus, David; Dinolt, George; Levin. E. Timothy, "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.

[7] J. Park, J. Choi, "Formal Security Policy Model for a Common Criteria Evaluation," ICACT2007,pp.277-281, Feb. 12-14, 2007.

[8] E. M. Clarke, J. M. Wing, "Formal Methods: State of the Art and Future Directions". ACM Comput. Surv. 28(4): 626-643 (1996).

[9] A.van Lamsweerde, "Formal specification: a roadmap", ICSE - Future of SE Track 2000: 147-159. 22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000. ACM 2000

[10] A. Bayley, H. Zhu, "Formal specification of the variants and behavioural features of design patterns". Journal of Systems and Software 83(2): 209-221, 2010.

[11] J. Julliand O. Kouchnarenko, Eds., B 2007," Formal Specification and Development in B", 7th International Conference of B Users, Besanc¸on, France, January 17-19, 2007, Proceedings, ser. Lecture Notes in Computer Science, vol. 4355. Springer, 2006.

[12] P. Frey, R. Radhakrishnan, H. Carter," A Formal Specification and Verification Framework for Time Warp-Based Parallel Simulation", IEEE Transaction on Software Engineering. Vol. 28, No. 1, January 2002

[13] S. Owre , N. Shankar," A Brief Overview of PVS",. Theorem Proving in Higher Order Logics Lecture Notes in Computer Science, , Volume 5170/2008, 22-27, 2008.

[14] Y. Bertot, "A Short Presentation of Coq", Theorem Proving in Higher Order Logics Lecture Notes in Computer Science, Volume 5170/2008, 12-16, 2008

[15] K. Slind ,M. Norrish, "A Brief Overview of HOL4", Theorem Proving in Higher Order Logics Lecture Notes in Computer Science, Volume 5170/2008, 28-32,2008.

[16] C.J. Date, H. Darwen, "Databases, Types, and Relational Model: The Third Manifesto", (3 rd edition); Addison-Wesley, 2007.

[17] A. Benabbou, S.Nait Bahloul, Y.Amghar, K. Rahmouni," Generation of an Orthogonal Object / Relational type system", Wotic 2009, WorkShop

international sur les Technologies de l'Information et de la Communication 24 – 25, Agadir, Maroc, Décembre 2009.

[18] A.Benabbou, S.Nait Bahloul, Y.Amghar, K. Rahmouni,"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.

[19] The Coq development team, The Coq proof assistant reference manual, LogiCal Project, 2012. [Online]. Available: http://coq.inria.fr

[20] J.H. Gallier, "on the Correspondence between Proofs and lambda-Terms", Technical Reports (CIS) Scholarly Commons university of University of Pennsylvania. 1993.

[21] M. Heine Sørensen, P. Urzyczyn," Lectures on the curry-howard isomorphism". University of Copenhagen 1998.

[22] Y. Bertot , P. Castéran , G. Huet , C. Paulin-Mohring," Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions" (Texts in Theoretical Computer Science). Springer; 1 edition, 2004.

[23] H. Barendregt," Lambda Calculi with Types", Handbook of Logic in Computer Science, Volume 1, Abramsky, Gabbay, Maibaum (Eds.), Clarendon 1992.

[24] A. Church," A Formulation of the Simple Theory of Types", The Journal of Symbolic Logic, 1940

[25] P.R. Halmos,"Naive Set Theory", D. Van Nostrand Company, Princeton, NJ, 1960. Reprinted, Springer-Verlag, New York, NY, 1974.

[26] C. Gonzalia,"Relations in Dependent type Theory". PhD Thesis, Chalmers University of Technology, 2006.

[27] M.Sozeau, N. Oury." 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 (2008) 278-293, August 18-21, 2008.

[28] P.Wadler, S.Blott. "How To Make ad-hoc Polymorphism Less ad hoc". In ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas, pages 60–76, 1989.

[29] C. Sinz, "System description: Ara - an automatic theorem prover for relation algebras". In Proc. CADE-17, 2000.

[30] C. McBride, J. McKinna, "Epigram: Practical Programming with Dependent Types". Advanced Functional Programming: 130-177, 2004.