Modeling Ontological Structures with Type Classes in Coq

Richard Dapoigny and Patrick Barlatier

LISTIC/Polytech'Annecy-Chambéry
University of Savoie, P.O. Box 80439, 74944 Annecy-le-vieux cedex, France
richard.dapoigny@univ-savoie.fr

Abstract. In the domain of ontology design as well as in Conceptual Modeling, representing universals is a challenging problem. Most approaches which have addressed this problem rely either on Description Logics (DLs) or on First Order Logic (FOL), but many difficulties remain especially about expressiveness. In mathematical logic and program checking, type theories have proved to be appealing but so far, they have not been applied in the formalization of ontologies. To bridge this gap, we present here the main capabilities of a theory for representing ontological structures in a dependently-typed framework which relies both on a constructive logic and on a functional type system. The usability of the theory is demonstrated with the Coq language which defines in a precise way what ontological primitives such as classes, relations, properties and meta-properties, are in terms of type classes.

1 Introduction

On the one hand, many researchers are striving for an ontologically well-founded representation language e.g., by adding new operators to Description Logics (DL) while preserving decidability, while on the other hand, mathematical and logical theories built on the Curry-Howard isomorphism have promoted the well-typedness as a foundational paradigm (above intuitionistic logic) leading to highly powerful theorem provers. Most modeling languages that have been proposed so far to express ontological constraints (or rules) are based on very simple meta-conceptualization as underlined in [21]. These languages offer appropriate structuring mechanisms such as classes, relationships and subsumption (subclass relations). However, in the representation of a formula, some structures have meaning whereas other do not make sense. This aspect requires "suitable ontological distinctions" understood as meta-properties of ontological structures as pointed out in [16] (e.g., the principles of identity or rigidity). In addition, the distinction of ontological meta-level categories such as types, kinds, roles, relations, etc., further make accurate and explicit the real-world semantics of the terms that are involved in domain representations. Not only an ontology is committed to represent knowledge of reality in a way that is independent of the different uses one can make of it, but it is intended to provide a certified and coherent map of a domain. All these constraints can be fulfilled within a highly expressive language built on a solid logical background. For that purpose in this paper, we propose a two-layered theory including a higher-order dependent type theory as a lower layer and an ontological layer as upper layer. This theory

H.D. Pfeiffer et al. (Eds.): ICCS 2013, LNAI 7735, pp. 135–152, 2013.

referred to, as K-DTT (Knowledge-based Dependent Type Theory) [2] is derived from [12] for the modeling of contexts. The logic in the lower layer operates on (names of) types whose meaning is constrained in the upper (ontological) layer.

Dependent types are based on the notion of indexed families of types and provide a high expressiveness since they can represent subset types, relations or constraints as typed structures. They will be exploited for representing knowledge in an elegant and secure way. This last aspect is analyzed in [8] where the authors investigate typing applied to reasoning languages of the Semantic Web and point out that dependent types ensure normalization. For example, type theory enjoys the property of subject reduction which ensures that no illegal term will appear during the execution of a well-typed query in a well-typed program. Alternatively, in [14], the authors have shown the ability of the type-theoretical approach to cope with scalability on the SUMO foundational ontology. Therefore, we introduce a a simple, coherent and decidable theory called K-DTT (Knowledge-based Dependent Type Theory) departing from the existing ones such as usual first-order logic theories. We will demonstrate with code fragments written in the support language Coq that the theory is able to satisfy most of all the constraints inherent in an expressive conceptual model.

2 Motivations for a Type-Theoretical Framework

In the conceptualization of information systems, Mylopoulos [32] has pointed out that the related language should be able to "formally represent the relevant knowledge". This assertion means that the conceptual language should (i) be expressive enough to represent the "relevant knowledge" and (ii) offer deduction capabilities to provide a valid model. It follows that there are three possible ways for increasing both the expressiveness and the soundness of a conceptual language (i) controlling the semantics of the conceptual language with a formal ontology (ii) using an expressive language e.g., an Object-Oriented language or (iii) using existing logic-based approaches such as Conceptual Graphs (CGs) or Description Logics. Let us review the basic features of these approaches.

Some authors such as [19] claim that a foundational ontology should allow to evaluate the "ontological correctness of a conceptual model" and to develop guidelines telling how the constructs of a conceptual modeling language should be used (e.g., association inclusion, specialization and redefinition). The author suggests that ontology adequacy should be a measure of the distance between the models produced by a modeling language and the real-world situations they are supposed to represent. To fulfill these constraints, the author has proposed an ontologically well-founded modeling language whose formal semantics is defined in a logical system as expressively as possible [20].

Object-Oriented languages [5], are the most significant formalisms for representing knowledge. They stem from frames which are seen as data structures that glue pieces of information together in their slots. In other words, OO languages are a computational implementation of frames. Classes correspond to frame descriptions while objects are identified to frame instances after filling in the slots. OO models offer two salient properties (i) the analogy between software models and physical models and (ii) the reusability of their components. The design of the program appears to be isomorphic to the components which result from the analysis of a given application with e.g.,

the UML tool. The central paradigm of OO languages is the notion of class. Classes encapsulate data (fields) and their properties (methods) in a single structure. classes are instantiated into objects which are designed to represent anything. Classes can be arranged into hierarchies using inheritance. A class can inherit behavior (i.e., data and properties) from another class called its superclass or parent class. Subsumption (i.e., polymorphism), is the ability to use a subclass where an object of its superclass is expected. However, many if not most, knowledgeable computing professionals recognize that the object-oriented paradigm is not the best one for every problem. In particular, despite some tentatives to add some logic to OO languages (e.g., F-logic [26]), the major weakness is that it lacks an expressive logical background for reasoning. For example, F-logic is very expressive but is generally undecidable.

The logical formalism of Conceptual Graphs (CGs)[43,7] originates in semantic networks and in the existential graphs of C.S. Peirce. It includes classes, relations, individuals and quantifiers with the purpose of providing a form humanly readable and computationally tractable. A CG has direct translation to the language of first order predicate logic, from which it takes its semantics. It results that CGs have the same expressing power as predicate logic. Labeled graphs with entities and relationships between them describe knowledge. CGs offer a significant advantage over concurrent formalisms, they are easily interpreted by end-users provided that they are not too complex. The model-theoretic semantics for the CGs is also specified in the ISO standard for Common Logic (CL). Common Logic includes the usual predicate-calculus notation for first-order logic. While CL semantics may represent entities of any type, it lacks ability for relating such entities to the internal structure of CL sentences. Reasoning in CGs relies on six canonical formation rules at the semantic level [43]. One can extend semantic operations with combinations of the rules, i.e., projection and maximal join. Graph operations such as projections (i.e., graph homomorphisms) are a major form of reasoning. A fundamental problem in simple CGs, i.e., deduction, starts with two simple CGs as input and searches whether a projection from the first to the second exists. The problem can be extended to a set of simple CGs (i.e., a knowledge base) and a simple CG representing a query (query answering). In querying simple conceptual graphs with negation, it has been shown that in the case of incomplete knowledge intuitionistic logic can be very attractive for capturing an answer to the query [31]. Finally, while CGs are efficient for representing natural language semantics, they do not support modality [44] and contexts are not taken into consideration when reasoning with CGs.

Alternatively, Description logics (DLs) refers to a family of knowledge representation formalisms that represent the relevant concepts of the domain (its terminology) together with properties of objects and individuals occurring in the domain. Semantically they are fragments of predicate logic, but their language is formed so that it would be enough for practical modeling purposes and also so that the logic would have good computational properties such as decidability. Very expressive DLs are likely to meet inference problems of high complexity, or to become undecidable.

We follow the idea of using an ontology for controlling the semantics of a conceptual model and will show in this paper how a well-founded model for the semantics, i.e., an ontologically correct conceptual model can be designed. Alternatively, the conceptual model must also be syntactically correct w.r.t. a set of rules and here we speak of a

well-formed model (this part has widely been explored in the literature and will not be addressed here). The analysis of ontological correctness boils down to design specifications. A specification has the same status as axioms of a mathematical theory, i.e., they can be proved. More precisely, one can prove that a specification is consistent (it does not include a contradiction), just as one can prove that the axioms of a theory are consistent. For that purpose, a strong theoretical framework together with a core foundational ontology are required. This ontology will rely on the classical dichotomy between universals and particulars. While some approaches exist, the selected framework should emphasize expressiveness while assuming a strong formal theory.

Most philosophers have only some knowledge about First-Order Logic (FOL) and as a consequence, most claims about universals, particulars, properties and relations have a FOL-based logical bias. If now we rather adopt a constructive logic, then the picture is different and if we associate this logic to a rich type system rooted in the lambda-calculus, then a different but coherent picture can be drawn. Therefore, we exploit here the Knowledge-based Dependent Type Theory (K-DTT) theory already introduced in [2]. The interesting point is that available tools exist (e.g., the Coq theorem prover) making more exploitable the theoretical picture. We will demonstrate with code fragments, written in the support language Coq, that the theory is able to satisfy most of all the constraints inherent in an expressive conceptual model. Using a higher-order polymorphic type theory provides a lot of benefits. First, higher-order is useful (i) to permit instances of categorization types to be types themselves, (ii) to abstract away from level distinctions and (iii) to directly support quantification over sets and general concepts. Second, the typed framework enjoys (i) the reduction of the search space by restricting the domains/ranges of functions, predicates and variables to subsets of the universe of discourse, (ii) a structured knowledge representation facilitating both assertions and class-hierarchies and (iii) the detection of type errors with well-typed formulas. Finally, using dependent types is crucial to offer a high expressiveness and to enforce semantic conditions. The approach of [24] combining an order-sorted logic with the ontological property classification is a first step in this direction. But we can do more by including a type system with a strong proof theory. Using an unified theory providing high expressiveness together with the ability to constrain semantics will give the knowledge engineer the tools to produce models with certain guaranteed properties in terms of ontological transparency, well-foundedness and re-usability. These aspects are possible since properties are treated on a par with meta-properties (see section 6.2). In addition, there are available theorem provers (e.g., Coq) which can be used to check the well-formedness of user-defined typed structures.

We assume a layered structure including a logical level subsumed by an ontological level. The lowest level is an intensional type theory based on previous works [9,36,46] giving rise to a computational theory and, at the highest level, to knowledge structures whose semantics relies on a hierarchy of concepts (e.g., the DOLCE hierarchy of particular categories) and the so-called meta-properties (as e.g., in Ontoclean). As a consequence, all the ontological classes introduced in the following must have corresponding structures satisfying typing mechanisms provided at the logical level. Unlike most representation languages, K-DTT provide constructs able to distinguish among terms having similar logical structure but different ontological meaning [18].

3 K-DTT: The Type-Theoretical Layer

The Type-Theoretical layer of K-DTT is both rooted in a constructive logic and on a typed λ-calculus using dependent types. Dependent type theories allow a type to be predicated on a value which makes them much more flexible and expressive than conventional type systems [30]. The constructive logic pre-supposes a logic centered on the concept of proof rather than truth and follows the Curry-Howard isomorphism [22] in which proving is "equivalent" to computing (or querying a database). Reasoning in K-DTT consists either in reducing types to their normal form or finding proofs for reduced types. In K-DTT, the type of a type is called a universe. Universes are partially ordered and are organized into an infinite hierarchy of predicative type universes $Type_i$ for data types together with an impredicative[1] universe noted $Prop$ for logic. This hierarchy follows a kind of cumulativity: $Prop \subseteq Type_0 \subseteq Type_1 \subseteq ....$ A universe is seen as a type that is closed under the type-forming operations of the calculus. Using the Curry-Howard isomorphism, terms of the type-theoretical layer can represent data structures as well as properties of these structures and proofs of these properties.

Definition 1. *Let Γ be a valid environment.*
A term T is called a type in Γ if $\Gamma \vdash T : U$ for some universe U.
A term M is called a proof object in Γ if $\Gamma \vdash M : T$ for some type T.

The underlying theory, i.e., the Calculus of Constructions with inductive types and universes (see e.g., [3]), has given rise to the Coq language[2] [10] which has recently promoted very powerful primitives such as Types Classes (TCs) [41,45] for describing data structures. TCs in Coq are a lot like type classes in Haskell, however Coq allows us to do better by specifying the rules inside TCs. Coq both combines a higher-order logic and a richly-typed functional programming language. All logical judgments in Coq are typing judgments such as $x : T$, where x is a variable and T, a term. The type-checker checks the correctness of proofs, that is, it checks using proof search that a data structure complies to its specification. The proof engine also provides an interactive proof assistant to build proofs using specific programs called tactics. The language of the Coq theorem prover consists in a sequence of declarations and definitions. A declaration associates a name with a specification. Specifications can be either logical propositions which reside in the universe $Prop$, mathematical collections which are in Set or abstract types which belong to a universe $Type_i$ with $i \in N$. The theory includes dependent types generalizing function spaces and Cartesian products.

Lemma 1. *[10] Let A, B, two types defined in the current context Γ such that $\Gamma \vdash s : \phi(x : A, B[x])$ where ϕ denotes a dependent type, then the universe of $\phi(x : A, B[x])$ is the maximum universe among the universes of A and B w.r.t. coercions.*

Dependent types give new power to TCs while types and values are unified. TCs have a structure derived from record types with fields, but they are more powerful by allowing parametric arguments, inheritance and multiple fields. For example if one wants to represent reflexive relations, we can introduce the `Reflexive` TC with:

[1] Impredicativity is a kind of conceptual circularity.
[2] The Coq language has reached a state where it is well usable as a research tool.

```
Class Reflexive {A} {R : relation A} :=
                reflexivity : forall x, R x x.
```

where `{ ... }` denotes implicit arguments, `A` stands for any type and `R : A → A → Prop` is a dependent type expressing mathematical relations. Notice that (i) the implicit type of variable `x` is automatically resolved in Coq to `x : A` and (ii) `relation` refers to the basic Coq library and complies with the above definition of `R`.

4 K-DTT: The Ontological Layer

To explain how to represent an ontology with K-DTT, we take the example of the DOLCE taxonomy of particulars [29,15] which does not classify universals and leaves room for conceptual choices about universal structures. The hierarchical taxonomy of particular categories will serve as a backbone, referred to as DOLCE backbone. The DOLCE backbone plus ontological commitments on appropriate structures expressed within type theory will form the ontological layer of the K-DTT theory. It can be used to express knowledge as long as the added features respect the core structures together with their logical constraints (see e.g., [11]). All data structures will be expressed either with operational TCs having a single field and returning a value in $Type_i$ or predicate classes returning a truth value in $Prop$.

4.1 Representing Ontological Classes

We follow the position adopted by most formal ontologies in computer science, i.e., that universals are general entities which are further refined in subcategories (e.g., relations) and that particulars are specific entities which exemplify universals but which cannot have themselves instances. K-DTT objects are equipped with meanings using the Curry-Howard isomorphism and assuming that any type (or typed structure) corresponds to a universal. Notice that "type" here is a mathematical notion and is not itself an object for ontological modeling. The fundamental ontological distinction between universals and particulars can be informally understood using the typing relation ":". Possible worlds which is a way of characterizing the distinction between descriptions (i.e., intensions) and particulars corresponding to the descriptions (i.e., extensions) is implicitly accounted for in K-DTT respectively with typed structures and sets of proof objects[3]. Properties and relations which correspond to predicates in a logical language are usually considered as universals. Terms of the ontological layer of K-DTT are built from the category $Universal$, (the highest universe in the lower layer) which includes the four basic categories (sub-universes in the lower layer), $\mathcal{C}$, $\mathcal{R}el$, $\mathcal{P}$ and $\mathcal{R}ol$ which stand respectively for concept types, relation types, property types and role types (see fig 1). The last category is not discussed here but more details can be found in [2].

The universe of concepts includes the set of universes for the foundational ontology $\{PT, AB, R, TR, T, PR, \ldots, STV, ST, PRO, \ldots, \}$ which refers to the DOLCE taxonomy. It classifies categories of particulars such as APO which stands for Agentive Physical Object. Each of these universes is closed under the type formation rule

[3] Proof objects are e.g., the result of queries on a database related to the ontology.

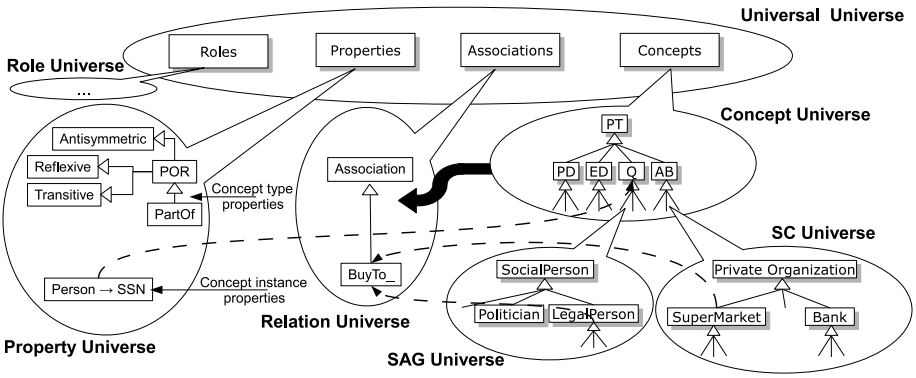

Fig. 1. Ontological categories in K-DTT

(lemma 1) and includes itself a set of sub-universes which can be defined in a domain ontology. All universes are partially ordered by subsumption formalized here by coercions [38] (e.g., ED is subsumed by PT, ...). In TCs, inheritance is implemented with (i) implicit arguments and (ii) the $:>$ operator. Concepts can be either primitive or compound. Primitive concepts are in line with the existence of "natural types", i.e., they can be identified as types in isolation (see e.g., [42]). Each category of particulars which belong to the DOLCE taxonomy is described with a TC where their position in the hierarchy is computed w.r.t. the coercion rules. All terms of the ontological layer must be well-formed.

Definition 2. *(Well-formed concepts) A term T is well-formed if for some formal ontology $\mathcal{O}$ providing the environment, we have either $\mathcal{O} \vdash T : C$ or $\mathcal{O} \vdash T : R$ or $\mathcal{O} \vdash T : P$ for some $C \in \mathcal{C}$, $R \in \mathcal{R}el$ and $P \in \mathcal{P}$.*

In the following, we often forget the environment for the sake of clarity, while any assertion will be relative to an explicit environment (e.g., a foundational ontology). For example, the judgment $LegalPerson : SAG \vdash x : LegalPerson$ where SAG denotes a Social AGent asserts that the variable x belongs to the concept $LegalPerson$ provided that it is a well-formed term, i.e., that $LegalPerson$ belongs to the universe SAG. A value for the variable x which can be e.g., $JohnDoe$, is called a proof object because it is both considered as an object (from the programming language side) and as a proof (from the logical side).

The relation of instantiation (:) between a universal and its instance corresponds here to type inhabitation. However, when this relation connects a type and its universe, it behaves like a Grothendieck universe [6] w.r.t. lemma 1. Unlike FOL-based ontologies, it describes in a natural and simple way the relation between universals and particulars without the need to introduce specialized ad hoc formulations. For example, in [40], the proposed theory of Is_a and $Part\_of$ is based on a relation of instantiation between an instance and a class and requires axioms for governing its use. In K-DTT, the relation of instantiation is already part of the theory and does not require any further axioms.

The equality between terms in the ontological layer is ascribed to be coherent w.r.t. the Leibniz equality of the lower layer (two types are logically identical iff they have

the same properties). It relates to the usual definition of the identity condition for an arbitrary property P, i.e., $P(x) \wedge P(y) \rightarrow (R(x,y) \leftrightarrow x = y)$ with a relation R satisfying this formula. This definition is carried out for any type in K-DTT since equality between types requires the Leibniz equality. The major reason is that identity can be uniquely characterized if the language is an higher-order language in which quantification over all properties is possible [34]. This property yields that Leibniz's Law, which is at the basis of identity in the lower layer of K-DTT is expressible in this language.

Relation types are rooted in the mathematical structure of a relation over a set A. The set is replaced by a general type $A : Type$ and rules can be defined over the basic TC $relation$ as described in section 3. It can be seen as a generalization of the work already presented in [11]. Specifications are a restriction of TCs with a first field for data structure and a second field for the property. Furthermore, fields of TCs can reside in any universe assuming the rule of lemma 1. Notice that proof objects for relation types are tuples and are closely related to lines of a table in a database. Universes of relation types are closed under type forming operations, i.e., Π-types and TCs constructs.

Property types reside in a sub-universe of $Universal$, which means that they are also universals. Properties of concept instances correspond to quality types (the DOLCE category Q) while properties of concept types correspond to meta-properties. Property types (concept or instance type properties) are divided into two sub-categories describing respectively mandatory properties (rigid properties) and possible properties (anti-rigid properties). This choice complies with the constraints of [17] arguing that anti-rigid properties cannot subsume rigid properties related to ontological distinctions in the current practice of conceptualization. While we agree with the introduction of three formal properties, i.e., identity, rigidity and dependence as proposed in [18], we do not share the view of properties as unary predicates since it relies on the restricted support of FOL (i.e., it has a logical bias).

4.2 Expressing Generalization

Generalization consists in deciding whether one category is "more general than" another one and is formalized by the subsumption relation "A subsumes B" which says that being a B logically implies being an A. The notion of subsumption has several readings, the more important ones being extensional and intensional [47,33]. There are some drawbacks to the extensional interpretation of subsumption because (i) determining whether the extension of one concept is included in the extension of another one is often undecidable and (ii) observing that two concepts have the same extension does not mean that they are identical.

In K-DTT, generalization and refinement are intensional and take advantage of two mechanisms (i) the simple mechanism of coercive subtyping [38,28] and (ii) TCs [41]. The hierarchy of generic concept types from DOLCE is isomorphic to a stratified hierarchy of universes in Coq (e.g., `PD` is (i) defined as a universe with the definition `:= Type` (ii) ordered with the typing assertion `: PT` and (iii) explicitly coerced with the parameter `u1`). In such a way, subsumption hierarchies can be designed provided that the coherence between coercion paths is preserved. This coherence is automatically checked in Coq (see e.g., [38]). The following fragment details this mechanism on some DOLCE categories showing e.g., that every perdurant (PD) and every endurant (ED) are also particulars (PT).

```
Definition PT           : Concept       := Type.
Definition PD           : PT            := Type.
Definition ED           : PT            := Type.
Parameter u1 : PD->PT.  Coercion u1 : PD>->PT.
Parameter u2 : ED->PT.  Coercion u2 : ED>->PT.
                        ...
```

The other mechanism creates inheritance hierarchies by refining TCs. Using the :> operator within a field of a TC means that any instance of the actual class is also an instance of the parent class. It results that TCs are a kind of bounded quantification where the subtyping relation needs not be internalized. When we represent higher and higher structures, TCs avoid the set of arguments growing as well. Predicate classes also support multiple inheritance which can be exploited in e.g., biomedical ontologies. Furthermore, they allow overlapping multiple inheritance which enable inherited structures to share components [45]. In such a way, very expressive hierarchies can be composed out of predicate classes and operational classes based on inheritance.

For example, we can add a TC for transitive relations to the operational TC of section 3 in a similar way. Then, a pre-order TC can inherit of these classes as follows:

```
Class Reflexive {A} {R : relation A} : Prop :=
          reflexivity : forall x, R x x.}
Class Transitive {A} {R : relation A} : Prop :=
          transitivity : forall x y z, R x y -> R y z -> R x z.}
Class PreOrder {A}{R:relation A} : Prop := {
          PreOrder_Reflexive   :> @Reflexive A R;
          PreOrder_Transitive  :> @Transitive A R }.
```

Here, the syntax :> declares each projection (i.e., each field) of the TC `PreOrder` as an instance of the respective TCs `Reflexive` and `Transitive`. It follows that each pre-order can be seen as a reflexive and as a transitive relation. This simple example highlights how multiple inheritance is implemented with implicit parameters.

5 Representing Relations

The strength of dependent types in type theory allows expressing relations as primitives of the language. The first consequence is that they are terms of the logic and can be involved in complex predicates. At the ontological level, relations are hierarchical (e.g., subsumption or part-of relations[4]) or non-hierarchical (e.g., domain relations). For the sake of simplicity, we only discuss here binary relations. Hierarchical relations are discussed in detail in section 6.2 since they require the specification of properties. Non-hierarchical relations denote tuples involving particulars and precisely correspond to instances of TCs having a first and a second field which detail their component types and a third one explaining how they are constructed and possibly other field(s) giving the additional properties they are subject to (see section 6.2). The basic TC `BinaryRel` defines the generic structure built out of a type A, another type B and a predicate type. For example, a domain relation type expressing persons which may suffer from a given

[4] We refer here to the part-of relation which is transitive by contrast with the partonomic relation which is usually not (see [11] for more explanations).

(or multiple) disease(s) inherits this basic structure through their implicit arguments. The specification `Class Person : APO` assumes that $Person$ belongs to the universe APO which itself is a sub-universe of $Concept$. Implicit respective parameters A and B are automatically resolved[5] with $Person$ and $Disease$ giving rise to the relation type `SufferFrom`:

```
Definition Disease         : PRO           := Type.
Parameter u20 : Disease->PRO. Coercion u20: Disease>->PRO.
Class Person               : APO           := { }.

Class BinaryRel {A B:PT}   : Association   := {
          BinaryRel_arg1   : A;
          BinaryRel_arg2   : B;
          BinaryRel_rule   : A->B->Prop}.
Class SufferFrom           : Association   := {
          SufferFrom_struc :> @BinaryRel Person Disease}.
```

Instances of $Person$ and $Disease$ correspond respectively to a set of persons and a set of diseases these persons are subject to. Notice that $Disease$ is defined as a universe since it cannot have direct instances. Each time an object having the type `@BinaryRel Person Disease` is introduced, Coq will try to construct an object of type `SufferFrom`. To clarify the meaning of relations in the typed framework, let us consider the appropriate table within a database (proof objects are items in the table):

$Person$	$Disease$
$John\_Doe$	$Influenza$ $AIDs$
$Mike\_Hammer$	$Herpes\_Simplex$
$Henry\_Mann$	$Rhinovirus$
$Franck\_Burch$	$Hepatitis\_A$ $nail\_infection$

The relation type $SuffersFrom$ belongs to the universe of (ontological) binary relations. A tuple $\langle John\_Doe, AIDs\rangle$ is a proof for the type $SuffersFrom$. Here proof objects are (constructively) obtained with requests to the database rather than by mathematical computation as usual.

Let us denote Tab and DB the respective contexts of the table related to the relation and the whole database. The set of persons[6] $Obj_{Tab}(Person) = \{John\_Doe,\ Mike\ Hammer, Henry\_Mann,\ Franck\_Burch\}$ relative to the table does not rule out the possibility to have other persons involved in other tables. We assume that DB is a valid context which contains at least every component of Tab. Using the weakening rule for contexts with $DB \vdash Person : Type$ and $Tab \vdash Person : Type$, we can assert $DB \vdash x : Person \supset Tab \vdash x : Person$[7]. The same reasoning holds

[5] Instance resolution is part of the Coq unifier.

[6] More formally, the set of proof objects.

[7] The symbol $\supset$ denotes the logical implication in higher-order logic.

for the type $Disease$. Then using the extensionality of computational equality between inhabited types [9], it follows that $Obj_{Tab}(Person) \subset Obj_{DB}(Person)$ and $Obj_{Tab}(Disease) \subset Obj_{DB}(Di\ sease)$. Furthermore, the diseases undergone by a person is a subset of the diseases undergone by any person within the table.

These definitions for binary relations can be extended to n-ary relations but this aspect will not be discussed here. The TC `BinaryRel` is general and applies at any abstraction level which is less than *PT*. Then Coq checks for the coherence of subsumption paths, for instance the sequence: `[u20; u15; u14] : Disease >-> PD`, `[u20; u15; u14; u1] : Disease >-> PT`, `[u20; u15] : Disease >-> STV`, `[u20] : Disease >-> PRO]` assumes that `Disease` is a `Concept` while providing the sequence of the coercion hops between these two terms. Notice that Coq checks for the coherence of these paths by avoiding multiple paths between two terms. This example illustrates how the types of the arguments are controlled throughout the refinement process and highlights the benefit of TCs for capturing knowledge.

6 Representing Properties

6.1 Concept Instances Properties

Concept instance properties or quality types can be attributed to things or predicated about them and then it can be said that objects exemplify quality types. In OO modeling, qualities (attributes) are embedded within the scope of a class while in CGs and DLs they rather have a relational flavor. In DOLCE, these qualities belong to a finite set of quality types (e.g., color, size, shape, etc.) and inhere in specific individuals. It results that two particulars cannot have the same properties (due to Leibniz equality) but they can have the same qualities, and at any time, a quality cannot exist unless the entity it inheres in also exists. A quality type is close to the "relational moment type" [20] which is also existentially dependent on other particulars. The inherence relation is isomorphic with type dependence on values. In K-DTT, the idea is to consider qualities as represented by TCs and then to attach them to other concepts through dependent arguments. Let us consider how a moment universal can be captured in K-DTT. We only consider here (for the sake of simplicity) an intrinsic moment which uniquely depends on a single particular. For example a person which has an attribute SSN (Social Security Number) can be conceptualized with first introducing two TCs with their appropriate coercions as follows:

```
Class SSN            : Q        := { SSN_Quale : nat}.
Class Person         : APO      := { ... }.
Class HasSSN         : Prop     := {
          SSN_Attr   : Person->SSN;
          SSN_mul    : SelectArity Person (card_0_1 Person)}.
```

where `SSN_Quale` is a natural number describing the SSN value. The TC `HasSSN` relates a person with its SSN through the type of the first field `Person->SSN`. The second field describe the arity of the SSN (we presuppose that a person has 0 or 1 SSN). Each arity has a lower and an upper value. Arity 1 is assumed if the field is a simple type declaration. Arity 0..1 can be represented with the predefined inductive type $option$. This type has two constructors, one referred to as $none$ says that there are no elements having the type whereas $Some\ t$ provides a term t having the type. Then,

any arity value greater than 1, i.e., the arity $1..n$ is easily represented with the list type. The resulting encoding formalizes these assumptions:

```
Inductive Arity (A:Type)   : Type  :=
             card_1        : Arity A
           | card_0_1      : Arity A
           | card_0_n      : Arity A.
Definition SelectArity (A:Type)(x:Arity A) : Type :=
           match x with
              | card_1     => A
              | card_0_1   => option A
              | card_0_n   => list A
           end.
```

With these definitions, it becomes easy to understand the second field in the `HasSSN` TC. It means that the type $person$ has an arity $0..1$, that is he has 0 or 1 SSN. If the SSN value is 0, then the related person has no SSN while any positive integer will provide the person as output. In summary, to express that any concept has a quality, a predicate TC (i) formalizes the dependency of the concept over the quality in a first field and (ii) formalizes the arity which inheres in the link person-SSN.

For each instance of `HasSSN`, one must be able to (i) construct an object (e.g., "33") having the type SSN and (ii) a proof that the cardinality $0..1$ has been selected.

```
Instance SSNPerson      : HasSSN     := {
        SSN_Attr John   := (Build_SSN 33);
        SSN_mul         := Multiplicity0_1 (Build_SSN 33) John }.
```

with:

```
Definition Multiplicity0_1 (s:SSN)(p:Person) : option Person :=
           match s with Build_SSN 0 => None
        |  Build_SSN s => Some p end.
```

On the one hand the dependent type allows to filter out unexpected values as arguments and unexpected quality types for the structural part of the property seen as a predicate TC, while on the other hand, inductive types provide a suitable mechanism for arity checking. Rigid and anti-rigid properties are useful if one can reason about them in a meta-schema as claimed in [17,18]. While basic constructors of type theory can be used for that purpose (see e.g., [2]), reasoning can be made easier by supplying a supplementary field to type classes, i.e., `Quality_rig`. Then this information can be used in a process which automate the control of domain ontologies resulting in well-formed hierarchies.

6.2 Concept Type Properties

Properties of concept types or meta-properties are described with rules, that is with predicate classes. For example to introduce partial order relations (POR), a widely-used concept in reasoning, one will extend the previous definitions for `Reflexive` and `Transitive` type classes as follows:

```
Class Antisymmetric { A } {R : relation A} : Prop :=
         antisymmetry : forall x y, R x y -> R y x -> x = y.
```

```
Class Irreflexive A R : relation A :=
        irreflexivity : forall x, R x x -> False.
Class Asymmetric {A}{R:relation A} := {
        Asym_Irreflexive                :> @Irreflexive A R;
        Asym_Antisymmetric              :> @Antisymmetric A R }.
Class POR { A }{R:relation A} : Prop    := {
        POR_Reflexive                   :> @Reflexive A R;
        POR_Antisymmetric               :> @Antisymmetric A R;
        POR_Transitive                  :> @Transitive A R }.
```

First, the coercions over class instances (e.g., `@Reflexive A R`) in the POR TC express multiple inheritance diagrams and second, the required types `A` and `relation A` are general and can be applied to any kind of relation. It follows that such a kind of inheritance diagram can be reused in any ontology-based application and give rise to modular hierarchies of axiomatic structural properties.

7 Constructing Inheritance Hierarchies

Usual theories about part-whole relations do not consider categories of the entity types involved in a part-whole relation and subsumption between these relations presuppose that their arguments are identical. The K-DTT theory is able to control both the types of each argument if required and inheritance between the type classes representing distinct relation types. This property extends the expected expressiveness of the theory beyond the usual power of ontology languages since it involves not only the predicates of relation types but also all arguments for these types. Let us consider mereotopological relations for endurants such as the has_3D property [25] which refers to both the endurant itself and the region (R) it occupies. The authors claim that the $contained\_in$ relation which is widely used in biology, involves both parthood and containment and can be captured with the first-order formula:

$$\forall x, y(contained\_in(x, y) \triangleq part\_of(x, y) \wedge R(x) \wedge R(y) \wedge \\ \exists z, w(has\_3D(z, x) \wedge has\_3D(w, y) \wedge ED(z) \wedge ED(w))$$

In K-DTT, this dual property is expressed using multiple inheritance. The inheritance diagram requires first the specification of a generic part-of relation. The inheritance with instances of type classes (i.e., `:> @PartOf R rr;`) avoids the introduction of a structural parthood in the taxonomy (see [25] for more details). Furthermore, simple inheritance with restriction on the type of argument will describe the `involved_in` relation, which relates two perdurants (see figure 2).

```
Definition ED   : PT                := Type;
Definition AB   : PT                := Type;
Definition R    : AB                := Type;
Class PartOf {A:PT}{rr:relation A} : Association := {
        PO_prop         :> @POR A rr }.
Class ProperPartOf {A:PT}{r:relation A} : Association := {
        PPO_propAs      :> @Asymmetric A r;
        PPO_propTr      :> @Transitive A r}.
```

```
Class ContainedIn {rr: relation R} : Association := {
        CIpartof_struct :> @PartOf R rr;
        CIregion_struct :> @BinaryRel R ED}.
Class InvolvedIn {rt: relation PD} : Association := {
        IIpartof_struct :> @PartOf PD rt}.
```

Then, assuming for example that we declare variable `regions : relation R`, we can prove with a tactic that any relation of type `ContainedIn` is transitive. The fact that we can apply transitivity to the arguments which belong to the class `ContainedIn`, means that it is propagated along the TC hierarchy until it reaches the TC `Transitive`. Then, applying twice the unification yields the result. Notice that tactics may be registered and reused which makes designer's task more easy by abstracting away the logical part.

```
Goal forall c:@ContainedIn regions, forall x y z:R, regions x y->
                                    regions y z -> regions x z.
Proof.
intros.
eapply transitivity.
eassumption.
assumption.
Qed.
```

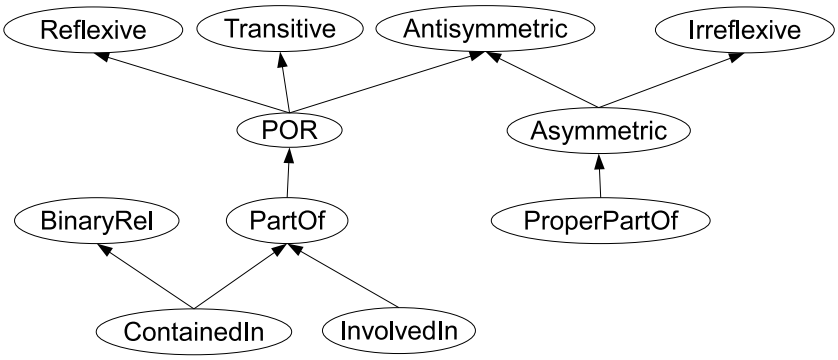

Fig. 2. An inheritance diagram in K-DTT

In a similar way, wider inheritance hierarchies can be investigated for describing expressive knowledge hierarchies in which type checking allow for proving well-formed ontologies. Automated proof-search could be improved by specifying ad'hoc requests on a database using parametrized requests (this aspect will be soon investigated).

8 Related Works

A subset of the CG theory called Prolog+CG is a Java implementation of Prolog where CGs are first-class datatypes on a par with terms [23]. It includes also Object oriented

extensions. This approach is based on typed hierarchy of concepts which is a lattice (The Sowa ontology) while our approach stems from the DOLCE foundational ontology and Ontoclean for the meta reasoning. Both approaches are using types but in Prolog+CG, typing requires multiple programming layers since Prolog does not naturally work with types. Instead, the K-DTT approach has a native typing system. While Prolog+CG has a Prolog core, nothing is said about negation whose handling is problematic. Due to the intuitionistic framework of K-DTT based on proof construction rather than discovering truth values, this problem is avoided. For that purpose, Coq has a lot of built-in tactics, and many more are available in libraries. Typically in Prolog, one expects the result(s) of the resolution and obtains a truth value while in Coq, one has to say what type has to be proved and obtains more information since a type is identified with the set of its proof objects. The creation of instances requires a primitive goal (CreateInstance) while Coq has a built-in notion of instanciation (:). The higher order capabilities of the type-theoretical layer are a crucial advantage for meta-reasoning.

Description logics are widely described and many tools exist which allow developers of to export their ontologies into a DL formalism. However, in DL-based biomedical ontologies, analysis have been investigated to check how their terminology complies with a basic set of ontological principles [4]. The authors have pointed out some inaccuracies of subsumption links, incomplete description and other conceptual ambiguities. Alternatively, in spatial reasoning [1], some limitations of OWL (and therefore of DL) are identified such as the difficulty of the language to represent the constraints which govern the coordinates of spatial objects in containment hierarchies. The comparison of datatype properties of individuals is the result of the lack of variables in DLs. Moreover, since DLs are conceptually oriented, they lack a rule-based reasoning mechanism such as the one found in logic programming (e.g., Horn clauses). Therefore, various approaches have been proposed for integrating logic programming and description logics such as [13,27,37]. Nevertheless, these reasoning techniques suffer from some limitations such as the restriction of DLs to "safe rules" and complicate the reasoning process by adding a translation mechanism between DLs and logic programming. More precisely, using these languages for practical applications raises several challenges [35]. The restriction to binary predicates in both SWRL and OWL is a first difficulty leading to violate safe rules for expressing the higher-order nature of the rules that have to be constructed. The main advantage of the present approach holds in the unified framework in which numerical values can be manipulated, rules can be applied and proved and multiple inheritance can be exploited while offering an expressive higher-order language.

9 Conclusion

The K-DTT theory is an attempt at constraining the semantics of knowledge representation based on expressive typed structures. While the logical part of the representation languages is neutral as concerns ontological choices, typing is not (e.g., the DOLCE backbone). K-DTT is a unifying theory both sufficiently expressive and logically founded together with a logic which supports different abstraction levels. We have (i) introduced the basic features of K-DTT and (ii) illustrated how modeling ontological knowledge can be checked in the Coq theorem prover. It is demonstrated that type

classes can model several non-trivial aspects of classes such as meta-level properties and multiple inheritance. TCs unify the two representations of relations, i.e., the logical view in which relations are predicates and the conceptual modeling view where a relation is seen as a set of tuples.

On the one hand, the present theory is more expressive than usual predicate logic in which it is neither possible to apply a function symbol to a proposition, nor to bind a variable except with a quantifier. In addition, the language of K-DTT is richer than FOL-based languages in allowing proofs to appear as parts of the propositions so that the propositions can express properties of proofs (and not only of individuals like in FOL). On the other hand, the distinction between usual Object Oriented programming and type theory relies on the ability of their representative computational structures to correctly express the semantics of ontological components. While their expressiveness is comparable, many aspects of object-oriented programming can be preserved in type theory since it unifies functional programming, component based programming, meta-programming (MDA), and logical verification (see [39] for more details). Further works include the realization of an interface between Coq and databases to collect proof objects with requests and a tool for proving well-formed ontologies.

References

1. Alia, I., Abdelmoty, A.I., Smart, P.D., Jones, C.B., Fu, G., Finch, D.: A critical evaluation of ontology languages for geographic information retrieval on the Internet. Journal of Visual Languages & Computing 16(4), 331–358 (2005)
2. Barlatier, P., Dapoigny, R.: A Type-Theoretical Approach for Ontologies: the Case of Roles. Applied Ontology 73, 311–356 (in press, 2012)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS series. Springer (2004)
4. Bodenreider, O., Smith, B., Kumar, A., Burgun, A.: Investigating subsumption in SNOMED CT: An exploration into large description logic-based biomedical terminologies. Artificial Intelligence in Medicine 39, 183–195 (2007)
5. Booch, G.: Object-Oriented Design with Applications. Benjamin Cummings, Redwood City (1991)
6. Bourbaki, N.: Univers, Séminaire de Géométrie Algébrique du Bois Marie Théorie des topos et cohomologie étale des schémas (SGA 4), 1. Lecture notes in mathematics, vol. 269, pp. 185–217. Springer (1972)
7. Chein, M., Mugnier, M.L., Simonet, G.: Nested graphs: a graph-based knowledge representation model with FOL semantics. In: Procs. of KR 1998, pp. 524–534. Morgan Kaufmann (1998)
8. Cirstea, H., Coquery, E., Drabent, W., Fages, F., Kirchner, C., Maluszynski, J., Wack, B.: Types for Web Rule Languages: a preliminary study. Technical report A04-R-560, PROTHEO - INRIA Lorraine - LORIA (2004)
9. Coquand, T., Huet, G.: The calculus of constructions. Information and Computation 76(2-3), 95–120 (1988)
10. Coq Development Team, The Coq Reference Manual, Version 8.3., INRIA, France (2010)
11. Dapoigny, R., Barlatier, P.: Towards Ontological Correctness of Part-whole Relations with Dependent Types. In: Procs. of the Sixth Int. Conference (FOIS 2010), pp. 45–58 (2010a)

12. Dapoigny, R., Barlatier, P.: Modeling Contexts with Dependent Types. Fundamenta Informaticae 104(4), 293–327 (2010b)
13. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. In: Proc. of Ninth Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2004), pp. 141–151. AAAI Press (2004)
14. Angelov, K., Enache, R.: Typeful Ontologies with Direct Multilingual Verbalization. In: Rosner, M., Fuchs, N.E. (eds.) CNL 2010. LNCS, vol. 7175, pp. 1–20. Springer, Heidelberg (2012)
15. Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., Schneider, L.: Sweetening Ontologies with DOLCE. In: Gómez-Pérez, A., Benjamins, V.R. (eds.) EKAW 2002. LNCS (LNAI), vol. 2473, pp. 166–181. Springer, Heidelberg (2002)
16. Guarino, N.: The Ontological Level. In: Casati, R., Smith, B., White, G. (eds.) Philosophy and the Cognitive Science, pp. 443–456. Holder-Pivhler-Tempsky (1994)
17. Guarino, N., Welty, C.: An Overview of OntoClean. In: Handbook on Ontologies, pp. 151–172 (2004)
18. Guarino, N.: The Ontological Level: Revisiting 30 Years of Knowledge Representation. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 52–67. Springer, Heidelberg (2009)
19. Guizzardi, G., Herre, H., Wagner, G.: On the General Ontological Foundations of Conceptual Modeling. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, pp. 65–78. Springer, Heidelberg (2002)
20. Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. University of Twente (Centre for Telematics and Information Technology) (2005)
21. Guizzardi, G., Masolo, C., Borgo, S.: In Defense of a Trope-Based Ontology for Conceptual Modeling: An Example with the Foundations of Attributes, Weak Entities and Datatypes. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 112–125. Springer, Heidelberg (2006)
22. Howard, W.A.: To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. The formulae-as-types notion of construction, pp. 479–490. Academic Press (1980)
23. Kabbaj, A., Janta-Polczynski, M.: From PROLOG++ to PROLOG+CG: A CG Object-Oriented Logic Programming Language, B. In: Ganter, B., Mineau, G.W. (eds.) ICCS 2000. LNCS (LNAI), vol. 1867, pp. 540–554. Springer, Heidelberg (2000)
24. Kaneiwa, K., Mizoguchi, R.: Ontological Knowledge Base Reasoning with Sort-Hierarchy and Rigidity. In: Procs. of KR 2004, pp. 278–288. AAAI Press (2004)
25. Keet, C.M., Artale, A.: Representing and reasoning over a taxonomy of part-whole relations. Applied Ontology 3(1-2), 91–110 (2008)
26. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM 42, 741–843 (1995)
27. Krötzsch, M., et al.: How to reason with OWL in a logic programming system. In: Procs. of RuleML 2006 (2006)
28. Luo, Z.: Coercive subtyping. Journal of Logic and Computation 9(1), 105–130 (1999)
29. Masolo, C., Borgo, S., Gangemi, A., Guarino, N., Oltramari, A.: Ontology Library (D18). Laboratory for Applied Ontology-ISTC-CNR (2003)
30. McKinna, J.: Why dependent types matter. In: Procs. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, vol. 41(1), p. 1 (2006)
31. Mugnier, M.L., Leclère, M.: On querying simple conceptual graphs with negation. Data & Knowledge engineering 60(3), 468–493 (2007)
32. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: Representing Knowledge About Information Systems. ACM Trans. on Information Systems 8(4), 325–362 (1990)

33. Napoli, A.: Subsumption and classification-based reasoning in object-based representations. In: Procs. of the 10th European Conference on Artificial Intelligence (ECAI 1992), pp. 425–429. John Wiley & Sons Ltd. (1992)
34. Noonan, H.: Identity. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy (2011), http://plato.stanford.edu/archives/win2011/entries/identity/
35. Pires, L.F., van Sinderen, M., Munthe-Kaas, E., Prokaev, S.M.H., Plas, D.J.: Techniques for describing and manipulating context information, Freeband/A MUSE D3.5v2.0, Lucent Technologies (2005)
36. Paulin-Mohring, C.: Inductive Definitions in the System Coq - Rules and Properties. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 328–345. Springer, Heidelberg (1993)
37. Rosati, R.: DL+log: Tight integration of description logics and disjunctive datalog. In: Proc. of Tenth Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006), pp. 68–78. AAAI Press (2006)
38. Saibi, A.: Typing algorithm in type theory with inheritance. In: Procs. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997), pp. 292–301. ACM Press (1997)
39. Setzer, A.: Object-Oriented Programming in Dependent Type Theory. In: Trends in Functional Programming, Intellect, vol. 7, pp. 91–108 (2007)
40. Smith, B., Rosse, C.: The Role of Foundational Relations in the Alignment of Biomedical Ontologies. In: Fieschi, M., et al. (eds.) MEDINFO 2004. IOS Press, Amsterdam (2004)
41. Sozeau, M., Oury, N.: First-Class Type Classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008)
42. Sowa, J.F.: Using a lexicon of canonical graphs in a semantic interpreter. Relational models of the lexicon, pp. 113–137. Cambridge University Press (1988)
43. Sowa, J.F.: Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks Cole Publishing Co., Pacific Grove (2000)
44. Sowa, J.F.: Conceptual Graphs. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, ch. 5, pp. 213–237. Elsevier (2008)
45. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. Mathematical Structures in Computer Science 21(4), 795–825 (2011)
46. Werner, B.: On the strength of proof-irrelevant type theories. Logical Methods in Computer Science 4(3) (2008)
47. Woods, W.A.: Understanding Subsumption and Taxonomy: a Framework for progress. In: Sowa, J. (ed.) Principles of Semantic Networks, pp. 45–94. Morgan Kaufmann (1991)