# Modeling Ontological Structures with Type Classes in Coq

Richard Dapoigny[1] Patrick Barlatier[1]

[1]LISTIC/Polytech'Savoie
University of Savoie, Annecy, (FRANCE)

Cite as: Dapoigny, R., Barlatier, P. Modeling Ontological Structures with Type Classes in Coq.

# Contents

# Some challenging problems in Conceptual Modeling

- The need for an ontologically well-founded representation language.
- The meta-conceptualization [Guizzardi06] of modeling languages is too simple.
- The usual structures of classes and relationships are not sufficient for expressing unambiguously reality.
- Meta-properties of ontological structures are not easily accounted for.
- The language should be able to support expressive axiomatic formalization of basic relations (*is_a*, *part_of*).

# Some challenging problems in Conceptual Modeling

## The idea: Why not to use Type-Theory?

- Not used so far for knowledge representation (except in NLP).
- The problem: few searchers in Conceptual Modeling are aware of Type Theory.
- The reason: it requires a strong commitment from users both in type theory and in conceptual modeling.

# Motivations for a Type-Theoretical Framework

- Based on Curry-Howard isomorphism: a typed $\lambda$-calculus is "equivalent" to a proof system in intuitionistic logic $\Rightarrow$ assessing a type is proving.
- Uses stratified universes (i.e., type of type) "à-la-Russell": a partial order hierarchy where each universe is closed under type-forming operations $\Rightarrow$ representing different abstraction levels is simplified.
- The underlying intuitionistic logic has been shown equiv. to S4 $\Rightarrow$ supports modality.
- Assumes Regular Word Assumption $\Rightarrow$ the set of proof objects in the database is a subset of the possible proof objects [Schur01].
- Uses Dependent types (product and sum types) $\Rightarrow$ the key for high-level expressiveness.

# Motivations for a Type-Theoretical Framework

## Type-theoretical basis

- Type Theories are based on the typed $\lambda$-calculus.
- We build on $CC^\omega$ [Coq88] and *CIC* [Paulin-Mohring96], the theoretical basic for the Coq language (http://coq.inria.fr/).
- Based on the concept of proof rather than truth.
- An infinite hierarchy of predicative type universes $Type_i$ for data types.
- An impredicative universe noted *Prop* for logic.
- Cumulativity: $Prop \subseteq Type_0 \subseteq Type_1 \subseteq ....$

# Motivations for a Type-Theoretical Framework

## Modality

- '$P$ is true iff $P$ is proved' (we can construct a proof for it) which implies identity between provability and $\Box$.
- Categorical judgments: empty context, e.g., $x : Patient$.
- Hypothetical judgments: under hypotheses, e.g., $\Gamma \vdash x : Patient$.
- Dependent justifications (expressed with dependent types) $\Rightarrow$ give the ability to express potential knowledge.

In the following, $\Gamma_{\mathcal{O}}$ will denote the context related to the ontology $\mathcal{O}$.

# K-DTT

## Description

- A two-layer theory.
  - The lower layer is type-theoretical (types and logic)
  - The higher layer provides ontological commitments which are interpreted in the lower layer.
- Ontological classes are expressed by terms which can be universes, types or compound terms.
- Particulars are understood in terms of proof objects.
- The relation of instantiation $:$ is already a primitive of the lower layer.

# The Ontological Layer

i. The context $\Gamma_{\mathcal{O}}$ is such that it includes all terms which are interpretations of universals (e.g., types or universes) in $[\![\mathcal{O}]\!]$,

ii. any particular $p \in \mathcal{P}$ is interpreted as the proof object $\Gamma_{\mathcal{O}} \vdash p : [\![U]\!]$, such that $\forall\, p \ : \ [\![U]\!]\ (\nexists\, p' \mid p' \ : \ [\![p]\!])$ with $[\![U]\!] : Type_0$, the type which interprets the universal $U$ related to $p$.

**A particular**

$$John\_Doe : LegalPerson$$

*LegalPerson* is well-formed iff $\Gamma_{\mathcal{O}} \vdash LegalPerson : [\![U]\!]$ for some universal $U$.

# The Ontological Layer

iii. any kind $K$ in $\mathcal{U}$ is interpreted as $\Gamma_{\mathcal{O}} \vdash [\![K]\!] : [\![U]\!]$ with $[\![U]\!] : Type_i$ with $i > 0$ and $[\![U]\!]$ which interprets the universal $U$.

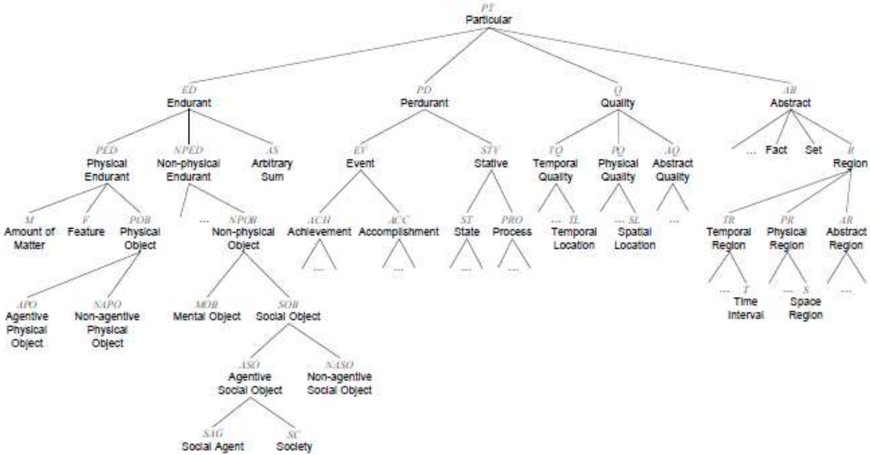
The hierarchical taxonomy of non-dependent kinds is the DOLCE hierarchy [Masolo03] and corresponds roughly to "natural types".

**A kind**

$$LegalPerson : SAG$$

# The Ontological Layer

## The DOLCE Backbone



# The Ontological Layer

iv. any property $P$ in $[\![\mathcal{O}]\!]$ may depend on a particular $x$ having the kind $K$ and is interpreted as the dependent kind $[\![P]\!][x : [\![K]\!]] : Q$,

**A property**

$$Color : Q$$

**A dependent property**

$$ColorOf(x : POB) : Q$$

*POB* refers to *Physical OBjects*. It means that all physical objects have a color which corresponds to "role types" [Sowa84].

# The Ontological Layer

v. any association $R$ relating universals $U_1$ and $U_2$ is interpreted as $[\![R]\!] : ([\![U_1]\!] \to [\![U_2]\!] \to Prop)$ with $\Sigma x : [\![U_1]\!] \,.\, \Sigma y : [\![U_2]\!] \,.\, [\![R]\!][x, y]$,

**A (binary) relation**

$$Relation : Kind \to Kind \to Prop$$

$$Class\ CarFrame\ : Association := \{...\}.$$

# The Ontological Layer

vi. any meta-property $S$ about a universal $U$ is the specification $S : ([\![U]\!] \rightarrow Prop)$ with $\Sigma x : [\![U]\!] \,.\, S[x]$.

**A specification over a relation**

Antisymmetric relations:

$$Antisymmetry : forall\ (X\ Y : Kind),$$
$$Relation\ X\ Y \rightarrow Relation\ Y\ X \rightarrow X = Y$$

# Using the Coq Language

- The Coq language has reached a state where it is well usable as a research tool [Bertot04].
- It is a sequence of declarations and definitions.
- It is designed such that evaluation always terminates (decidability).
- The type-checker checks using proof search that a data structure complies to its specification.
- Very powerful primitives: Types Classes (TCs) [Sozeau08,Spitters11] for describing data structures.

# Primitives

- Instantiation : $x : A \Rightarrow$ all properties of $A$ are available in $x$.
- Equality : Leibniz equality $\Rightarrow$ types are equal iff they have the same properties.
- Subsumption $\Rightarrow$ coercive subtyping [Saibi97] (coherence checked).

## Coercive subtyping in Coq

$$\frac{\Gamma \vdash M \ : \ A \quad c_{AA'} : A \ \leqslant \ A'}{\Gamma \vdash (c_{AA'} M) \ : \ A'} \ (\mathrm{Coerce})$$

$$\frac{f : T_1 \to T_2 \quad c_1 : U_1 \leqslant T_1 \quad c_2 : T_2 \leqslant U_2}{x : U_1 \vdash (c_2(f(c_1 x))) : U_1 \to U_2} \ (\mathrm{Coerce-Pi})$$

- Type Classes (TCs): extension of sum types.

# Subsumption

The Object-Oriented view:

- If a class $A$ is subclass of class $B$:
  - Every instance of $A$ is also an instance of $B$
  - Values of properties of $B$ are inherited by instances of $A$
- There are many examples where the use of subclass-of relation can be incorrect in subtle ways.

Another way is to separate properties from classes e.g., in DL and CGs $\Rightarrow$ we share this view.

# Kinds

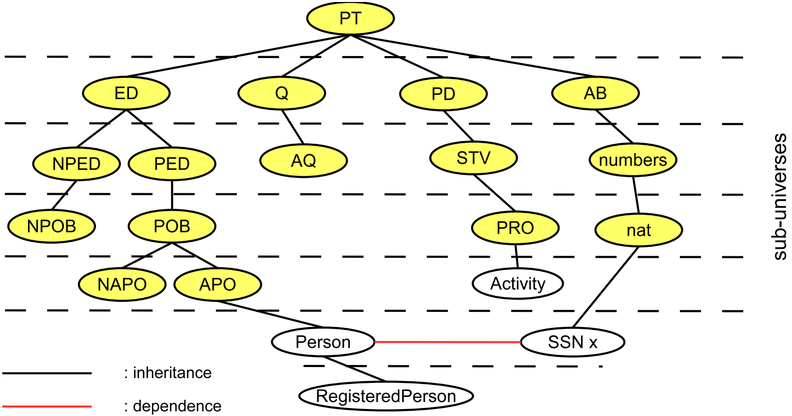
- Universes are available in Coq, but hardly manipulable
- We define a universe *Universal* and two sub-universes for *Kinds* and *Associations*:

```
Definition Universal                          := Type.
Definition Kind          :  Universal := Type.
Definition Association   :  Universal := Type.
```

- Two parametric universes are created for a foundational and a domain ontology:

```
Definition OntoCore (c:Type)                  := Type.
Definition OntoDomain (c:Type)                := Type.
```

# Kinds


PT
ED
Q
PD
AB
NPED
PED
AQ
STV
numbers
NPOB
POB
PRO
nat
NAPO
APO
Activity
Person
SSN x
RegisteredPerson
: inheritance
: dependence
sub-universes

# Properties

Some code example: a Kind *Person* and a property *SSN* parameterized with an object of type *Person*.

## A property: SSN

```
Structure Person              :  OntoDomain APO    := {
                                 Personsub :> APO }.
Structure SSN (x:Person)      :  OntoDomain AQ     := {
                                 SSN_Quale :> nat}.

Structure RegisteredPerson :  OntoDomain Person := {
                                 Rpst :> Person;
                                 RpProp :  SSN Rpst}.
```

It follows that *RegisteredPerson* is a subset of Person, i.e., persons constrained by the property "having a SSN" (anti-rigid property).

# Properties

## Cardinality Constraints (Core Ontology)

```
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.
```

# Properties

## Using Cardinality Constraints (Domain Ontology)

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

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.
```

# The results

## Some Benefits

- Kinds and properties are separate $\Rightarrow$ portability between domains (unlike OO frameworks).
- We have inheritance as in OO frameworks (all properties of the super-class are available in the class).
- We have coercive subtyping (an object of a super-class can be used instead of an object of a class).
- Mixing Structures (for coercive subtyping) and their powerful version (Type Classes) for inheritance $\Rightarrow$ modularity.
- Basic cardinality constraints can be defined $\Rightarrow$ conceptual modeling.

# Context

- The focus here is on binary associations (relations).
- The more interesting case is hierarchical (formal relations) such as *is_a* and *PartOf*.
- The first one is already part of K-DTT, then the focus is on *PartOf* relations.

# *PartOf* Relations

## The basic part-of relation

```
Parameter Relation :  (OntoCore Kind) → (OntoCore Kind) → Prop.

Class PartOf {X Y :OntoCore Kind}      :  Prop    := {
          PartOf_s          :> Relation X Y;
          PartOf_pred       :> POR}.
```

Any instance of a part-of relation both inherits the structure of a generic binary relation and the properties of a partial order relation.

# *PartOf* Relations

## A (sub) part-of relation

```
Class CarFrame  :  Association  := {
                a               :  Frame;
                b               :  Car;
                CarFrame_prop   :> @ProperPartOf Frame Car}.
```

Some domain associations may inherits from part-whole relations.

# Expressing meta-properties

- Properties of kind types (or meta-properties) are described with rules.
- Example: the partial order (meta) property POR.

# Expressing meta-properties

## POR

```
Class Antisymmetric :  Prop            :=
                   Antisymmetry :  forall (x y:OntoCore Kind),
                   Relation x y → Relation y x → x = y.
Class Reflexive    :  Prop            :=
                   Reflexivity :  forall x:OntoCore Kind,
                   Relation x x.
Class Transitive   :  Prop            :=
                   Transitivity :  forall x y z:OntoCore Kind,
                   Relation x y → Relation y z → Relation x z.

Class POR          :  Prop            := {
        POR_Refl                       :> Reflexive;
        POR_Antisym                    :> Antisymmetric;
        POR_Trans                      :> Transitive}.
```

# Expressing meta-properties

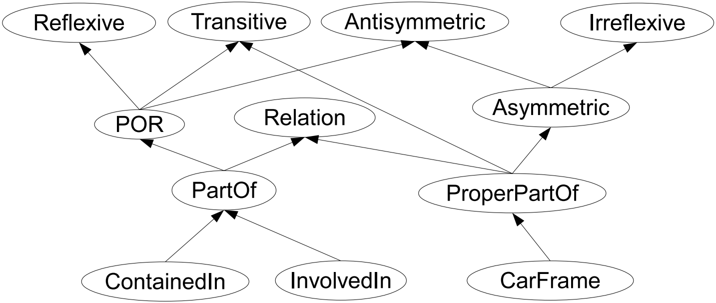
## Inheritance of part-whole relations

```
Class ProperPartOf {X Y :OntoCore Kind} :  Prop := {
        PPO_s            :> Relation X Y;
        PPO_Asym         :> Asymmetric;
        PPO_Trans        :> Transitive}.

Class ContainedIn { X Y: OntoCore Kind} :  Prop := {
        ContainedIn_prop:> @PartOf X Y}.
```

# Expressing meta-properties

We have the ability to build part-whole hierarchies [Keet08].



# Going a Step Further

- Tactics can be either achieved interactively with the user or built with the language LTac.
- More than one hundred tactics are available in Coq.
- Tactics can be the building block of high-level reasoning.

## Incompatible relations

Suppose that we want to prove that a relation, e.g., having the type `@ProperPartOf Frame Car` cannot be a part-of relation which swaps the arguments.

## ProperPartOf is not a PartOf

```
Lemma PPO_not_PO : forall c:@ProperPartOf Frame Car,
                                  ~(@PartOf Car Frame).
Proof.
intros.
decompose record c.
unfold not.
intro p.
decompose record p.
eapply Asymmetry.
exact PPO_s0.
exact PartOf_s0.
Qed.
```

# In summary

- K-DTT is a unifying theory both sufficiently expressive and logically founded together with a logic which supports different abstraction levels.
- Its implementation in Coq is able to constrain the semantics of knowledge representation based on expressive typed structures.
- The higher order capabilities of the type-theoretical layer are a crucial advantage for meta-reasoning.
- TCs 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 and the conceptual modeling view.
- Many aspects of OO programming can be preserved in type theory since it unifies functional programming, component based programming, meta-programming (MDA), and logical verification (see [Setzer07]).

# Thematic map

Goal:
research for a unified language for conceptual modeling and logic

## Application of type theory

Modeling contexts
(RIA 2008)

Modeling contexts
(Fundamenta Informaticae 2010)

Modeling situations
(Information Fusion 2013)

Ontological Modeling
& Part-whole relations
(ER 2009)

Modeling Part-whole relations with specifications
(FOIS 2010)

Modeling Roles
(Applied Ontology 2012)

Modeling Ontological Structures with TCs in Coq
(ICCS 2013)

Representing & reasoning with formal relations
(J. of Logic & Computation)

**Coq**

# Perspectives

- All development holds in 3 steps:
  - Step 1 $\Rightarrow$ Theoretical development see [Dapoigny Barlatier09] (ER09), [Dapoigny Barlatier10a] (FOIS10), [Dapoigny Barlatier10b] (Fundamenta Informaticae), [Dapoigny Barlatier13] (Information Fusion).
  - Step 2 $\Rightarrow$ Implementation in Coq [Barlatier Dapoigny12] (Applied Ontology), this paper and other paper submitted.
  - Step 3 $\Rightarrow$ Design of an interface tool to abstract away the formal theory to be achieved ...
- The future:
  - investigating more aspects in Coq,
  - building the interface.

# Thanks for your attention.