ON FORMALIZATION OF CATEGORY THEORY

カテゴリー理論の形式化について

by

Takahisa Mohri

毛利 貴久

A Senior Thesis

卒業論文

Submitted to

the Department of Information Science

the Faculty of Science

the University of Tokyo

on February 21, 1995

in Partial Fulfillment of the Requirements

for the Degree of Bachelor of Science

Thesis Supervisor: Masami Hagiya　萩谷 昌己

Title: Associate Professor of Information Science

ABSTRACT

Category theory is important in several areas of computer science, such as semantics and implementation of functional and imperative programming languages, the design of programs, typing, et. On the other hand, in reserches called formalized mathematics, various areas of mathematics are formalized and formal proofs are checked on computers. Category theory is also one of the objects of these researches.

For category theory, there are some formalizations based on set theory or type theory(Martin-Löf style, ECC).

In this paper, we compare the advantage and disadvantage of these formalizations. In particular, in advanced category theory including notions of functor category,set-valued functor, etc., the ability to deal with higher-order concepts and notions of sets is necessary. From this point of view, as an example, we attempt to prove Yoneda's Lemma based on each formalization.

Then we discuss which is the best method of formalization of category theory, and actually implement it on a proof system. Finally we examine required functions of a proof system for the formalization of category theory and problems on the implementation. .

論 文 要 旨

カテゴリー理論は、関数型および命令的プログラミング言語の意味論と実装、プログラムの設計、型付けなど、計算機科学のさまざまな分野において重要である。一方、数学の諸分野を形式化し、計算機上で形式的な証明を検証する形式的数学の研究が進められており、カテゴリー理論もその対象となっている。

カテゴリー理論に対しては、集合論、型理論 (Martin-Löf式, ECC) にもとづく形式化が行なわれている。

本論文では、それぞれの形式化の長所やを欠点の比較を行なう。特に、より進んだカテゴリー理論では、 functor category, set-valued functor 等を定義するために、高階性、集合の概念を扱う能力が必要である。この観点から各方法を比較するため、例として、 Yoneda's Lemma を各々の形式化のもとで証明することを試みる。

それをふまえてカテゴリー理論の形式化の最良の方法を論じ、さらに実際の証明系の上に実装する。カテゴリー理論の形式化のために証明系に必要な機能や実装上の問題点を検討する。

Acknowledgements

I would like to express my thanks to people who provided information for this work. I am especially thankful to Masami Hagiya for his much precious suggestions.

Table of Contents

Chapter 1

Introduction

1.1 Background on formalization of mathematics

The idea for checking logical derivations automatically has been suggested since long ago. It can be traced back to Pascal, Leibniz, and even to Ramon Llull. Recently, by using computers as the above automatic device, various systems which assist for doing mathematics on the machines are developed. Among the better known are: Nuprl[11], Mizar[33], LEGO[32, 24], Coq[15], Isabelle[31], HOL[16], and many others[21, 22]. The project developing each system has various specific goals. However, one of the common feature is that formal proofs are written by the human being, checked on the computer. Thus, formalization of mathematics is required for the machine to check proofs.

With respect to which theory the system is based on, these are classified into Mizar and the others. Mizar is based on set theory. Basically, mathematical notions are based on set theory. Therefore many definitions therefore are naively formalized in Mizar.

The others are based on type theory. Due to the "propositions-as-types" principle, mathematical objects and logical objects are uniformly treated in type theory. From the question of what the computational content of logic is, this principle is discovered by H.B.Curry[13], W.Howard[19], de Bruijn[14] and Lauchli[23]. Associated to the principle, proofs are interpreted as inhabitants of these types and there exist judgements of inhabitation and equality. They forms a typed λ-calculus or type theory. It is called Martin-Löf's Intuitionistic type theory[29, 28]. Martin-Löf's type theory contains concepts of the dependent type and universes. The dependent types are consists of the dependent function types and the dependent products. The dependent function type is the type of functions whose range type can depends on the input. In the inpendent product, the type of the second member of a pair can depend on the value of the first. They, respectively, originate in the indexed product of a family of sets and the indexed disjoint sum of a family of sets in set theory. Thus they are called Π-type and Σ-type(or strong sum type) respectively. The type structure hierarchy of this type theory is the hierarchy of universes. A universe is closed under all the type-forming operations except formation of a universe of higher or equal level. The transfinite hierarchy is cumulative; that is a universe is in the next universe and every element of a universe

is also an element of the next universes. Universes are themselves types. The hierarchy makes the theory meta-theoretic.

The Calculus of Constructions[12] and the Extend Calculus of Construction[25, 26] are important type systems discovered recently. *The Calculus of Constructions*,CC is a higher-order formalism for constructive proofs which the "corresponds" between propositions and types is used as a strong paradigm for computer science. In it, every proof is formalized as a typed λ-expression, expressing its associated algorithm and computing this λ-term is corresponds to cut-elimination in natural deduction style. These λ-terms are pure strongly normalizable; that is, all computations terminates. *The Extend Calculus of Constructions*,ECC is a higher-order calculus which combines CC[12] and Martin-Löf's type theory with universes. In the hierarchy of universes, the type $Prop$ of all propositions is in the lowest type universe $Type_0$ and every inhabitant of $Prop$ is also an inhabitant of $Type_0$ and a universe is closed under Pi- and Σ- abstraction. By means of distinguishing between propositions and types, CC and ECC depart significantly from the Curry-Howard tradition. ECC has an $\omega-\mathbf{Set}$ (realizability) model, and $\omega-\mathbf{Set}$ is a category.

1.2 Background on category theory

The main connection between programming language theory and category theory is the fact that both are "theory of functions". "morphism" in category theory generalizes the set-theoretical function in a very broad sense. Therefore, category theory is important in several areas of computer science, such as semantics and implementation of functional and imperative programming languages, the design of programs, typing, et.

Similar to other theory, category theory is introduced based on set theory. The followings are naive definitions of basic notions in category theory:

- **Category** A **category C** is
  - a collection $Ob_{\mathbf{C}}$ of **objects**
  - a collection $Mor_{\mathbf{C}}$ of **morphisms(arrows)**
  - two operations **dom, cod** assigning to each arrow f two objects respectively called **domain(source)** and **codmain (target)** of f
  - an operation **id** assigning to each object b a morphism id_b (the **identity** of b) such that $dom(id_b) = cod(id_b) = b$
  - an operation "$\circ$" (**composition**) assigning to each pair f, g of arrows with $dom(f) = cod(g)$ an arrow $f \circ g$ such that $dom(f \circ g) = dom(g), cod(f \circ g) = cod(f)$
  - identity and compositions, moreover, must satisfy the following conditions:

    identity law : for any arrows f, g such that $cod(f) = b = dom(g)$

    * $id_b \circ f = f$

* $f \circ id_b = f$

associative law : for any arrows f, g, h such that $dom(f) = cod(g)$ and $dom(g) = cod(h)$

* $(f \circ g) \circ h = f \circ (g \circ h)$

In category theory, a morphism f with douce a and target$b$ is denoted by $f : a \to b$ and $\mathbf{C}$[a,b] denotes the collection of all morphisms f such that $f : a \to b$.

- **Functor** Let $\mathbf{C}$ and $\mathbf{D}$ be categories. A **functor** $\mathbf{F} : \mathbf{C} \to \mathbf{D}$ is a pair of operations $\mathbf{F}_{ob} : Ob_{\mathbf{C}} \to Ob_{\mathbf{D}}, \mathbf{F}_{mor} : Mor_{\mathbf{C}} \to Mor_{\mathbf{D}}$ such that, for each $f : a \to b, g : b \to c$ in $\mathbf{C}$,
  - $\mathbf{F}_{mor}(f) : \mathbf{F}_{ob}(a) \to \mathbf{F}_{ob}(b)$
  - $\mathbf{F}_{mor}(g \circ f) = \mathbf{F}_{mor}(g) \circ \mathbf{F}_{mor}(f)$
  - $\mathbf{F}_{mor}(id_a) = id_{\mathbf{F}_{ob}(a)}$.
- **Natural Transformation** Let $\mathbf{F}, \mathbf{G} : \mathbf{C} \to \mathbf{D}$ be functors. Then $\tau : \mathbf{F} \to \mathbf{G}$ is a **natural transformation** from $\mathbf{F}$ to $\mathbf{G}$ iff p
  - $\forall a \in Ob_{\mathbf{C}} \quad \tau_a \in \mathbf{D}[\mathbf{F}(a), \mathbf{F}(b)]$;
  - $\forall f \in \mathbf{C}[a, b] \quad \tau_b \circ \mathbf{F}(f) = \mathbf{G}(f) \circ \tau_a$.

1.3 Motivation

As mentioned above, category theory is important in computer science. And category theory is a large theory in a sense that it has a meta-theoretic aspect. For example,$\omega-$**Set** (realizability) model of ECC is a category. Therefore, by trying to formalize category theory in a system, one can discuss an ability of the system. In this paper, we formalize category theory and compare them with existing formalizations based on set-theory, Martin-Löf type theory and ECC.

The naive definition of category theory is based on set theory. In the set-theoretic system, it will be formalized straightforward. But, in the type-theoretic system, treatment of notion of set is a matter of importance. In process of formalizing the category **Set** of all sets, we discuss it.

1.4 Existing formalization of category theory

Broadly speaking, there are three exiting formalizations for category theory. These are based on set theory, and type theory(Martin-Löf style, ECC).

- **Formalization on Mizar system**

  Set-theoretic formalization is implemented on Mizar system[33]. Mizar has adopted Tarski-Grothendieck axiomatization out of various formalizations of set theory. Tarski-Grothendieck

set theory **TG** is stronger than Gödel-Bernyes set theory **GB**, i.e. notion of set in **TG** is larger than notion of set in **BG**.

In the formalization[6, 7, 9, 10, 5, 34, 8], notions of category, functor, natural transformation, functor category, product in category, product of category, subcategory, opposite category, category **Ens**, Hom-functor are defined and related propositions proved.

For each non-empty set of sets V, $\mathbf{Ens}_V$ is defined to the category with the objects of all sets $X \in V$, morphisms of all mapping from X into Y, with the usual composition of mappings.

- **Formalization on Nuprl system**

  It is implemented by Altucher and Panangaden in [1] on Nuprl system [11]. The type system of Nuprl is based on Martin-Löf style type theory, and has subtypes and some set-theoretic constructors. In the Nuprl type system, equality is given for universes. By cumulativity, equality in a universe is the restriction of equality in the next universe. By closedness of universes, equality in a universe is the restriction of type equality to members of the universe. Equality on Pi-type is extensional, and equality on Σ-type is pairwise. Equality in a subtype of a type A is just the restriction of equality in A to the subtype.

  In [1], notions of category, functor, natural transformation are defined to prove the adjoint functor theorem.

- **Formalization on LEGO system**

  It is implemented on LEGO system[32, 24] by Aczel to define algebras like monoid and group based on category theory. In it, based on ECC type theory, notion of category are defined with notion of setoid and the category SET constructed. In type theory, as shown in [4], a set is defined as a type T with a total equivalence relation on T or as a type T with a partial equivalence relation on T. The former is exactly the definition of setoid.

  The formalization with notion of setoid was translated to Coq system[15] based on the Calculus of Inductive Construction by Huet. The Calculus of Inductive Construction is extension of CC with primitive inductive definition(see [27, 30]). Inductive definition is introduced to define free algebras without the use of impredecative encoding. The calculus allow inductive types and inductive predicate to express recursive type. In the translation, Σ-types are converted to inductive types.

1.5 Objectives and Contribution

At first, based on the existing three formalizations, basic notions in category theory are formalized in each theory. And we construct functor category and formalize proofs of Yoneda lemma and its application by using these formalized notions.

Let **C,D** be categories, functor category from **C** to **D**, **Funct(C,D)** is the category whose objects are functors and whose morphisms are natural transformations. **Nat**[F,G] is the abbreviation used instead of **Funct(C,D)**[F,G]. Comparing formalizations of this Higher-order concepts in each theory, we discuss the level of mathematical objects in a sense.

Yoneda's lemma states that for each functor $\mathbf{K} : \mathbf{C} \rightarrow \mathbf{Set}$, $Nat(\mathbf{C}[r,], \mathbf{K})$ and $\mathbf{K}(r)$ are isomorphic. Here, **C** is a locally small category and $\mathbf{C}[r,]$ is hom-functor and category **Set** is the category of all sets. A category **C** is locally small when for all $a, b \in Ob_{\mathbf{C}}$, $\mathbf{C}[a, b]$ is a set. If **C** is locally small and $a \in Ob_{\mathbf{C}}$, the hom-functor $\mathbf{C}[a,] : \mathbf{C} \rightarrow \mathbf{Set}$ maps $b \in Ob_{\mathbf{C}}$ to $\mathbf{C}[a, b]$. In the above sentences, set means set in **GB**. Notion of set is twice here and **Set** is very important in category theory. We formalize proofs of the lemma to discuss "notion of set" in each theory.

Advanced these formalizations in this paper, more properties and notions are able to be defined as future work. In particular, Yoneda's lemma helps to formalize properties associated to topos. As another advanced work, algebra such as monoid and group are defined on category theory.

1.6 Outline

In Chapter 2, we introduce the above existing formalizations and define basic notions in category theory (category, functor, natural transformation) based on each formalizations.

To check the ability to deal with higher-order concepts and notion of set, we formalize functor category and Yoneda's lemma in Chapter 3 and 4 respectively.

In Chapter 5, we compare these formalizations with respect to level of type universe hierarchy, equality on morphisms, and category **Set**.

In Chapter 6, we conclude the comparison and describe future works. ‘

Chapter 2

Preliminaries

2.1 Introduction of three existing formalizations

There exist three formalizations for category theory. For each formalization, the adopted system and its underlying theory and contents of the formalization are surveyed in subsection.

2.1.1 Formalization on Mizar system

The original goal of the project MIZAR was to design and implement software environment to assist the process of preparing mathematical papers. And it can be seen as attempt to develop software environment for writing traditional papers, where classical logic and set theory form the basis of all future developments.

Mizar system[33] is based on Tarski-Grothendieck set theory,**TG**. Notion of set in **TG** includes notion of set in Gödel-Bernyes set theory **GB** and larger then it. In **TG**, sets at a higher level in a sense can be defined then in **GB**.

The followings are existing formalizations of notions in category theory on Mizar system.

- Category and functor are formalized in [6].
- Subcategory and products of categories are done in [7].
- Product in category is done in [9].
- Cartesian category are done in [10].
- Natural transformation and functor category are formalized in [34].
- Opposite category and contravariant functor are done in [8].
- Category Ens V and Hom-functors are done in [5].

After the notions are defined, proofs of associated theorems are formalized. Contents of these formalizations are described latter section and chapters.

2.1.2 Formalization on Nuprl system

The Nuprl project is a step toward realizing the goal which the Bourbaki effort manifests; that is to provide for mathematics a uniform notation for discourse and a facility for creating an encyclopedia of results in this notation.

The type system of Nuprl is based on Martin-Löf's Intuitionistic type theories[29, 28]. It has dependent function type and dependent product and some set-theoretic constructors such as subtype, disjoint union and quotient type. There exist equality and judgement in the Martin Löf style type theory.

Equality in a universe U_i is the restriction of equality in the next universe U_{i+1} because the universe are cumulative. Equality in a universe is the restriction of type equality to members of the universe because the universes are closures. Equality on dependent function type x:A–>B is extensional; that is, f = g in x:A–>B iff for all a in A f(a) = g(a) in B[a/x]. Equality on dependent product x:A#B is pairwise; that is, <a,b> = <a',b'> in x:A#B iff a = a' in A and b = b' in B[a/x]. Equality in subtype {x:A|B} is just the restriction of equality in type A to {x:A|B}.

In [1], basic notions in category theory are formalized to prove the adjoint functor theorem as the below. Here, subtyping and other set constructors is not used and `=` is a definitional equality in Martin-Löf's type theory.

```
Category == Obj:U2
          # Mor:(Obj # Obj)->U1
          # Id:(o1:Obj->Mor(o1,o1))
          # o:(o1:Obj->o2:Obj->o3:Obj
                ->Mor(o1,o2)->Mor(o2,o3)->Mor(o1,o3)
          # forall A,B:Obj.forall f:Mor(A,B).
                f o Id(A) A,A,B = f in Mor(A,B)
          # forall A,B:Obj.forall g:Mor(B,A).
                Id(B) o g B,A,A = g in Mor(B,A)
          # forall A,B,C,D:Obj.
            forall f:Mor(C,D).forall g:Mor(B,C).forall h:Mor(A,B).
        ((f o g B,C,D) o h A,B,D) =
        (f o (g o h A,B,C) A,C,D)           in Mor(A,D)
```

In the above definition, `U1` is the universe at the lowest level and U2 is the next. The collection of objects is in a higher universe(`U2`) than the collection of morphisms.

```
Functor(A,B) ==
        f1:|A| -> |B|
      # f2:(o1:|A| -> o2:|B| -> Hom(A)(o1,o2) -> Hom(B)(f1(o1),f2(o2)))
```

```
# forall x,y,z:|A|. forall f:Hom(A)(y,z).forall g:Hom(A)(x,y).
  f2(x)(z)(f o(A) g) = f2(y)(z)(f) o(B) f2(x)(y)(g)
                       in Hom(B)(f1(x),f1(z))
# forall x:|A|. f2(x)(x)(Id(A)(x)) = Id(B)(f1(x))
                                     in Hom(B)(f1(x),f1(x))
```

For a category A, |A| stands for the type of objects in A. The notation o is a function that takes a category and returns the composition of that category. (f o(A) g) means composition in the category A. Id and Hom are similar to it. The additional arguments that occur in the definition of composition are suppressed. But the definition typed into Nuprl system should have these arguments. Hom is a function that takes a category and extracts the morphism function, the latter is a function that takes two objects and returns the type of morphism between those objects.

```
Nat_Trans(C,D,F,G) ==
        n: (o1:|C| -> Hom(D)(F(o1),G(o1)))
     #  forall a,b:|C|.forall f:Hom(C)(a,b). G(f) o n(a) = n(b) o F(f)
```

To make precise the notion of natural transformation is part of the original motivation in [1]

2.1.3 Formalization on LEGO system

For formalizing theorems and developing proofs, LEGO[32, 24] supports various related type systems. LEGO implements the theories including the followings.

- LF:Edinburgh Logical Framework[18]
- PCC:Pure Calculus of Constructions[12]
- CC:(Generalized) Calculus of Constructions[12]
- XCC:Extend Calculus of Constructions[25, 26]

They are related in the above order, from weaker to stronger. PCC is what we call ECC and XCC is what we call ECC. Details of the type theories and their relationships and differences are in [32]. These theories can be extended with new inductive types. LEGO supports universe polymorphism (or typical ambiguity)[17, 20]. One can omit the subscripts n of $Type_n$.

The formalizations on LEGO by Aczel is based on ECC. And algebras like monoid and group are defined on it. In it, category is formalized with setoid, a type T with a total equivalence relation on T. It is the type of total sets in [4], which defines basic mathematical notions in type theory.

The formalization was translated to Coq system[15] by Huet. Coq is based on the Calculus of Inductive Construction, which is extension of CC with primitive inductive definition[27, 30]. Inductive definition is introduced to define free algebras without the use of impredecative encoding. To define a inductive types, one must specify its constructors which determine the inhabitants of

the type and the induction principle attached to this type. The followings are translation by G.Huet.

```
Inductive Definition Setoid : U
  = Setoid_intro : (S:U)(R:(Rel S))(Equiv S R) -> Setoid.

Section Maps.
Variables A,B: Setoid.
Definition Map_law = [f:(elem A)->(elem B)]
    (x,y:(elem A))(equal A x y) -> (equal B (f x) (f y)).
Inductive Definition Map : U =
    Map_intro : (f:(elem A)->(elem B))(p:(Map_law f))Map.
Definition ap = [m:Map](<(elem A)->(elem B)>Match m with
         [f:(elem A)->(elem B)][p:(Map_law f)]f).
Definition ext = [f,g:Map]
         (x:(elem A))(equal B (ap f x) (ap g x)).
Lemma Equiv_map_eq : (Equiv Map ext).
Definition Map_setoid = (Setoid_intro Map ext Equiv_map_eq).
End Maps.

Definition ap2 =
  [A,B,C:Setoid][f:(elem (Map_setoid A (Map_setoid B C)))][a:(elem A)]
  (ap B C (ap A (Map_setoid B C) f a)).

Section cat.
Variable Ob:U. (* Objects *)
Variable H:Ob->Ob->Setoid. (* Hom Setoid *)
Definition Comp_Type =
  (a,b,c:Ob)(elem (Map_setoid (H a b) (Map_setoid (H b c) (H a c)))).
Variable o:Comp_Type.
Definition comp = [a,b,c:Ob](ap2 (H a b) (H b c) (H a c) (o a b c)).
Definition Assoc_law = (a,b,c,d:Ob)
      (f:(elem (H a b)))(g:(elem (H b c)))(h:(elem (H c d)))
      (equal (H a d) (comp a b d f (comp b c d g h))
                     (comp a c d (comp a b c f g) h)).
Definition Id_Type = (a:Ob)(elem (H a a)).
```

```
Variable id:Id_Type.
Definition Idl_law = (a,b:Ob)
   (f:(elem (H a b))) (equal (H a b) (comp a a b (id a) f) f).
Definition Idr_law = (a,b:Ob)
   (f:(elem (H b a))) (equal (H b a) f (comp b a a f (id a))).
End cat.

Inductive Definition Category : U =
 Cat_intro : (Ob:U)(Hom:Ob->Ob->Setoid)
             (o:(Comp_Type Ob Hom))(i:(Id_Type Ob Hom))
             (Assoc_law Ob Hom o) -> (Idl_law Ob Hom o i)
                                  -> (Idr_law Ob Hom o i) -> Category.

Lemma comp_set : (Comp_Type Setoid Map_setoid).
Lemma assoc_set : (Assoc_law Setoid Map_setoid comp_set).
Lemma id_set : (A:Setoid)(Map A A).
Lemma idl_set : (Idl_law Setoid Map_setoid comp_set id_set).
Lemma idr_set : (Idr_law Setoid Map_setoid comp_set id_set).
Definition SET : Category =
    (Cat_intro Setoid Map_setoid comp_set id_set assoc_set idl_set idr_set).
```

In the translations, Σ-types are converted to inductive types in the definitions of setoid, map, and category. Here, `U` is a sort of all types `Type`, `Rel` returns binary operation, `Equiv` returns a proof that the relation is a equivalent relation. `elem` and `equal` are selectors of setoid. `elem` returns elements and `equal` returns equivalent relation.

2.2 Formalizations of basic notions

In this section and latter chapters, we formalize each notion in three way as the followings and compare them.

- S : Based on existing formalization on Mizar
- T1: Based on existing formalization on Nuprl
- T2: Based on existing formalization on LEGO

Their underlying theories are the below

- S : Tarski-Grothendieck set theory
- T1: Martin-Löf style type theory with subtyping

- T2: ECC type theory

Mizar syntax are used for the formalization S. To comparing two type theory, we use common syntax for T1 and T2.

The full defintions for S,T1 and T2 are in the Appendix A, B, C respectively.

2.2.1 Category

The following is definition of categories in the formalization S from [6].

```
struct CatStr
   ≪ Objects,Morphisms -> non empty set,
     Dom,Cod -> (Function of the Morphisms, the Objects),
     Comp -> (PartFunc of [:the Morphisms, the Morphisms :],the Morphisms),
     Id -> Function of the Objects, the Morphisms
   ≫;
definition
 mode Category-like->CatStr means
      (for f,g being Element of the Morphisms of it holds
         [g,f] ∈ dom(the Comp of it) iff (the Dom of it).g=(the Cod of it).f)
   &  (for f,g being Element of the Morphisms of it
         st (the Dom of it).g=(the Cod of it).f holds
         (the Dom of it).((the Comp of it).[g,f]) = (the Dom of it).f &
         (the Cod of it).((the Comp of it).[g,f]) = (the Cod of it).g)
    & (for f,g,h being Element of the Morphisms of it
         st (the Dom of it).h = (the Cod of it).g &
            (the Dom of it).g = (the Cod of it).f
        holds (the Comp of it).[h,(the Comp of it).[g,f]]
            = (the Comp of it).[(the Comp of it).[h,g],f] )
    & (for b being Element of the Objects of it holds
         (the Dom of it).((the Id of it).b) = b &
         (the Cod of it).((the Id of it).b) = b &
         (for f being Element of the Morphisms of it st (the Cod of it).f = b
           holds (the Comp of it).[(the Id of it).b,f] = f ) &
         (for g being Element of the Morphisms of it st (the Dom of it).g = b
           holds (the Comp of it).[g,(the Id of it).b] = g ) );
end;
```

In the above, category structures are defined as 6-tuples consist of

- objects, morphisms: a non-empty set

- a dom-map, a cod-map: a function from the morphisms into the objects
- a composition: a partial function from [:the morphisms, the morphisms:] to the morphisms. (The notation [: • , • :] means a direct product)
- an id-map: a function from the morphisms into the objects.

And categories are defined as subsets of the structures satisfies the following propositions.

- The domain of its composition includes all composable pairs of morphisms.
- The proposition on dom and cod of the composed morphisms.
- The associative law.
- The identity laws.

It is a naive definition of category.

We translated the definition on Nuprl in the previous section as in the formalization T1.

```
{
  Ob:Type(i)
  H:Ob → Ob → Type(j)
  Comp_Type = Πa:Ob.Πb:Ob.Πc:Ob.(Hom a b) → (Hom b c) → (Hom a c)
  Id_Type = Πa:Ob.H a a
  o:Comp_Type
  Assoc_law = Πa:Ob.Πb:Ob.Πc:Ob.Πd:Ob.
              Πf:(Hom c d).Πg:(Hom b c).Πh:(Hom a b).
                (o a c d f (o a b c g h)) = (o a b d (o b c d f g) h)
                                          ∈ (Hom a d))
  id:Id_Type
  Idl_law = (Πa:Ob.Πb:Ob.Πg:(Hom a b).f = (o a a b f (Id a)) ∈ (Hom a b))
  Idr_law = (Πa:Ob.Πb:Ob.Πg:(Hom b a).(o b a a (Id a) g) = g ∈ (Hom b a))
}
Cat(i,j) = ΣOb:Type(i).ΣHom:Ob → Ob → Setoid(j).
           Σo:Comp_Type Ob Hom.Σi:Id_Type Ob Hom.
           (Assoc_law Ob Hom o) × (Idl_law Ob Hom o i) × (Idr_law Ob Hom o i)
```

$=$, $\in$ are definitional equality and judgement in Martin-Löf's type theory. We use i,j as subscript of type universe and the type-theoretic definitions in this paper are universe polymorphic. In the definition, we assume disjointness of Hom-set and define the type `Cat(i,j)` of categories as a dependent products of 7 types:

- Ob: a type of all objects, the type is in Type(i),

- Hom: a type of the function returns Hom[a,b], Hom[a,b] is in Type(j),
- o: a type of composition-operation,
- Id: a type of identity-operation,
- a the identity laws,
- the associative law.

We translated the definition on Coq in the previous section as the formalization T2.

Because definitional equality does not exist in ECC, we define general setoids, and setoids of maps from setoid to setoid at first.

```
Setoid(i) = ΣS:Type(i).ΣR:Rel S.Equiv S R
{
  A:Setoid(i)
  B:Setoid(i)
  Map_law = λf:elem A → elem B.Πx:elem A.Πy:elem A.
              equal A x y → equal B (f x) (f y)
  Map = Σf:elem A → elem B.Map_law f
  ap = λm:Map.π1 m
  ext = λf:Map.λg:Map.
        Πx:elem A.equal B (ap f x) (ap g x)
  Map_setoid = (Map, ext, Equiv_map_eq)
}
```

The type `Map` is the dependent product of the type of functions from setoid to setoid and the proposition that states the functions are set-theoretic functions. `Map_setoid` is the setoid of Map with extensional equality on Map.

```
{
  Ob:Type(i)
  H:Ob → Ob → Setoid(j)
  Comp_Type = Πa:Ob.Πb:Ob.Πc:Ob.
              elem (Map_setoid (H b c) (Map_setoid (H a b) (H a c)))
  Id_Type = Πa:Ob.elem (H a a)
  o:Comp_Type
  comp = λa:Ob.λb:Ob.λc:Ob.ap2 (H b c) (H a b) (H a c) (o a b c)
  Assoc_law = Πa:Ob.Πb:Ob.Πc:Ob.Πd:Ob.
              Πf:elem (H c d).Πg:elem (H b c).Πh:elem (H a b).
              equal (H a d) (comp a c d f (comp a b c g h))
```

```
                        (comp a b d (comp b c d f g) h)
  id:Id_Type
  Idl_law = Πa:Ob.Πb:Ob.Πf:elem (H a b).
    equal (H a b) f (comp a a b f (id a))
  Idr_law = Πa:Ob.Πb:Ob.Πf:elem (H b a).
    equal (H b a) (comp b a a (id a) f) f
}

Cat(i,j) = ΣOb:Type(i).ΣHom:Ob → Ob → Setoid(j).
           Σo:Comp_Type Ob Hom.Σi:Id_Type Ob Hom.
           (Assoc_law Ob Hom o) × (Idl_law Ob Hom o i) × (Idr_law Ob Hom o i)
```

In the definition, we assume disjointness of Hom-set and define the type `Cat(i,j)` of categories as a dependent products of 7 types similar to T1. `ap2 A B C` is a function takes as arguments a function $f : A \to B \to C$ and an elements a of A and returns $f(a) : B \to C$.

2.2.2 Functor

The following is definition of functors in the formalization S from [6].

```
::Functor
reserve C,D for Category;
definition let C,D;
 mode Functor of C,D -> Function of the Morphisms of C,the Morphisms of D
   means
    ( for c being Element of the Objects of C
       ex d being Element of the Objects of D
        st it.((the Id of C).c) = (the Id of D).d )
    & ( for f being Element of the Morphisms of C holds
         it.((the Id of C).((the Dom of C).f)) =
            (the Id of D).((the Dom of D).(it.f)) &
         it.((the Id of C).((the Cod of C).f)) =
            (the Id of D).((the Cod of D).(it.f)) )
    & ( for f,g being Element of the Morphisms of C
          st [g,f] ∈ dom(the Comp of C)
         holds it.((the Comp of C).[g,f]) = (the Comp of D).[it.g,it.f] );
end;
```

The definition of functors is different to the naive definition in Chapter 1. In this definition, functors from category C to D are defined as subsets of functions from $Mor_{\mathbf{C}}$ to $Mor_{\mathbf{D}}$. This is

correspond to $\mathbf{F}_{mor}$ in Chapter 1.

$\mathbf{F}_{ob}$ are determined by `Obj`. The function `Obj` takes a functor $\mathbf{F}_{mor}$ and returns object functor $\mathbf{F}_{ob}$ of the functor $\mathbf{F}$.

```
:: Object Function of a Functor
definition let C,D;
 let F be Function of the Morphisms of C,the Morphisms of D;
 assume
 for c being Element of the Objects of C
    ex d being Element of the Objects of D
     st F.((the Id of C).c) = (the Id of D).d;
 func Obj(F) -> Function of the Objects of C,the Objects of D means
  for c being Element of the Objects of C
    for d being Element of the Objects of D
     st F.((the Id of C).c) = (the Id of D).d holds it.c = d;
end;
```

The following is a definition of functors translated as in the formalization T1.

```
{
  C:Cat(i,j)
  D:Cat(k,l)
  {
    F1:Ob C → Ob D
    F2:Πa:Ob C.Πb:Ob C.Hom C a b → Hom D (F1 a) (F1 b)
    Comp_law = Πa:Ob C.Πb:Ob C.Πc:Ob C.
               Πf:Hom C b c.Πg:Hom C a b.
               ((F2 a c (o C a b c f g)) =
                (o D (F1 a) (F1 b) (F1 c) (F2 b c f) (F2 a b g))
                 ∈ (Hom D (F1 a) (F1 c)))
    Id_law = (Πa:Ob C.(F2 a a (id C a)) = (id D (F1 a)) ∈ (Hom D (F1 a) (F1 a)))
  }
  Functor(i,j,k,l) = ΣF1:Ob C → Ob D.
                     ΣF2:Πa:Ob C.Πb:Ob C.Hom C a b → Hom D (F1 a) (F1 b)
                     (Comp_law F1 F2) × (Id_law F1 F2)
}
```

`Ob,Hom,o,id` are selectors of a category. They returns objects, Hom-function, composition and identity respectively. In the definition, the type `Functor(i,j,k,l)` of functors from `Cat(i,j)` to `Cat(k,l)` are defined as the dependent products of 4 types:

- F1: the type of $\mathbf{F}_{ob}$, operation from $Ob_{\mathbf{C}}$ to $Ob_{\mathbf{D}}$
- F2: the type of $\mathbf{F}_{mor}$, operation from $Mor_{\mathbf{C}}$ to $Mor_{\mathbf{D}}$
- the proposition on the functor to composed pair,
- the proposition on the functor to identity.

Similar to the definition in T1, we define the type `Functor(i,j,k,l)` as the formalization T2 in the followings.

```
{
  C:Cat(i,j)
  D:Cat(k,l)
  {
    F1:Ob C → Ob D
    F2:Πa:Ob C.Πb:Ob C.elem (Hom C a b) → elem (Hom D (F1 a) (F1 b))
    compC = comp (Ob C) (Hom C) (o D)
    compD = comp (Ob D) (Hom D) (o D)
    Comp_law = Πa:Ob C.Πb:Ob C.Πc:Ob C.
               Πf:elem (Hom C b c).Πg:elem (Hom C a b).
               equal (Hom D (F1 a) (F1 c))
                 (F2 a c (compC a b c f g))
                 (compD (F1 a) (F1 b) (F1 c)(F2 b c f) (F2 a b g))
    Id_law = Πa:Ob C.equal (Hom D (F1 a) (F1 a)) (F2 a a (id C a)) (id D (F1 a))
  }
  Functor(i,j,k,l) = ΣF1:Ob C → Ob D.
                     ΣF2:Πa:Ob C.Πb:Ob C.elem (Hom C a b) → elem (Hom D (F1 a) (F1 b)).
                     (Comp_law F1 F2) × (Id_law F1 F2)
}
```

2.2.3 Natural transformation

In the formalization S from [34], natural transformations are defined from the definition of transformations in the following.

```
::Transformations
reserve A,B for Category,
        F,F1,F2 for Functor of A,B;
definition let A,B,F1,F2;
 pred F1 is_transformable_to F2 means
 for a being Object of A holds Hom(F1.a,F2.a) <> ∅
```

```
end;
definition let A,B,F1,F2;
 assume  F1 is_transformable_to F2;
 mode transformation of F1,F2 ->
         Function of the Objects of A, the Morphisms of B means
 for a being Object of A holds it.a is Morphism of F1.a,F2.a;
end;
```

In the above, transformations are defined. By using them, natural transformations are defined as the following.

```
:: Natural transformations
definition let A,B,F1,F2;
 pred F1 is_naturally_transformable_to F2 means
 F1 is_transformable_to F2 &
 ex t being transformation of F1,F2 st
  for a,b being Object of A st Hom(a,b) <> ∅
   for f being Morphism of a,b holds t.b • F1.f = F2.f • t.a;
end;
definition let A,B,F1,F2;
 assume F1 is_naturally_transformable_to F2;
 mode natural_transformation of F1,F2 -> transformation of F1,F2 means
  for a,b being Object of A st Hom(a,b) <> ∅ for f being Morphism of a,b holds it.b •
F1.f = F2.f • it.a;
end;
```

We translated the definition of natural transformations in the previous section as in the formalization T1.

```
{
  C:Cat(i,j)
  D:Cat(k,l)
  F:Functor(i,j,k,l) C D
  G:Functor(i,j,k,l) C D
  F1 = π1 F
  F2 = π2 F
  G1 = π1 G
  G2 = π2 G
  NatTrans = Σtau:Πa:Ob C.Hom D (F1 a) (G1 a).
             Πa:Ob C.Πb:Ob C.Πf:Hom C a b.
```

```
                ((o (F1 a) (G1 a) (G1 b) (G2 a b f) (tau a)) =
                 (o (F1 a) (F1 b) (G1 b) (tau b) (F2 a b f))
                                              ∈ (Hom D (F1 a) (G1 b)))
}
```

In the definition, the type `Functor(i,j,k,l)` of functors from `Cat(i,j)` to `Cat(k,l)` are defined as the dependent products of 2 types:

- tau: the type of functions from a in $Ob_{\mathbf{C}}$ to Hom[F(a),G(a)],
- the naturality.

Similar to the definition in T1, we define the type `Functor(i,j,k,l)` as formalization T2 in the followings.

```
{
  C:Cat(i,j)
  D:Cat(k,l)
  F:Functor(i,j,k,l) C D
  G:Functor(i,j,k,l) C D
  F1 = π1 F
  F2 = π2 F
  G1 = π1 G
  G2 = π2 G
  o = comp (Ob D) (Hom D) (o D)
  NatTrans = Σtau:Πa:Ob0 C.elem (Hom D (F1 a) (G1 a)).
             Πa:Ob0 C.Πb:Ob0 C.Πf:elem (Hom C a b).
             equal (Hom D (F1 a) (G1 b))
               (o (F1 a) (G1 a) (G1 b) (G2 a b f) (tau a))
               (o (F1 a) (F1 b) (G1 b) (tau b) (F2 a b f))
}
```

Chapter 3

Functor category

Let **C,D** be categories, category of functors from **C** to **D**, **Funct(C,D)** is the category whose objects are functors and whose morphisms are natural transformations from **F** to **G**.

In the formalization S from [34], functor categories are defined as the following.

```
:: Functor category
reserve A,B for Category;
definition let A,B;
 mode NatTrans-DOMAIN of A,B -> non empty set means
 for x being Any holds x ∈ t implies
  ex F1,F2 being Functor of A,B, t being natural_transformation of F1,F2
   st x = [[F1,F2],t] & F1 is_naturally_transformable_to F2;
end;
definition let A,B;
 func NatTrans(A,B) -> NatTrans-DOMAIN of A,B means
 for x being Any holds x ∈ it iff
  ex F1,F2 being Functor of A,B, t being natural_transformation of F1,F2
   st x = [[F1,F2],t] & F1 is_naturally_transformable_to F2;
end;
definition let A,B;
 func Functors(A,B) -> strict Category means
 the Objects of it = Funct(A,B) &
   the Morphisms of it = NatTrans(A,B) &
   (for f being Morphism of it holds dom f = f‘1‘1 & cod f = f‘1‘2) &
   (for f,g being Morphism of it st dom g = cod f
     holds [g,f] ∈ dom the Comp of it) &
   (for f,g being Morphism of it st [g,f] ∈ dom (the Comp of it)
     ex F,F1,F2,t,t1 st f = [[F,F1],t] & g = [[F1,F2],t1] &
```

```
      (the Comp of it).[g,f] = [[F,F2],t1∘t]) &
   for a being Object of it, F st F = a holds id a = [[F,F],id F];
end;
```

In the above, NatTrans-DOMAIN is defined as the set of all natural transformations and NatTrans(A,B) is natural formations between functors from A to B. t1∘t means the (vertical) composition of natural transformations t1 and t.

For the formalization T1, we define the type Funct(C,D) of functor category from C to D as the followings.

```
{
  C:Cat(i,j)
  D:Cat(k,l)
  Functs = Functor(i,j,k,l) C D
  {
    F:Functs
    G:Functs
    NatTr = {π1 (NatTrans C D F G)|π2 (NatTrans C D F G)}
  }
  comp_nat = λF1:Functs.λF2:Functs.λF3:Functs.
             λt1:NatTr F2 F3.λt2:NatTr F1 F2.
             λa:C.o D (t1 a) (t2 a)
  id_nat = λF:Functs.λa:C.id D (π1 F a)
  assoc_nat = λF1:Functs.λF2:Functs.λF3:Functs.
              λt1:NatTr F3 F4.λt2:NatTr F2 F3.λt3:NatTr F1 F2.
              pr_assoc D (π1 F1 a) (π1 F2 a) (π1 F3 a) (π1 F4 a)
                 (t1 a) (t2 a) (t3 a)
  idl_nat = λF1:Functs.λF2:Functs.λt:NatTr F1 F2.
    pr_idl D (π1 F1 a) (π1 F2 a) (t a)
  idr_nat = λF1:Functs.λF2:Functs.λt:NatTr F2 F1.
    pr_idr D (π1 F1 a) (π1 F2 a) (t a)
  Funct = (Functor(i,j,k,l) C D, NatTr, comp_nat, id_nat,
             assoc_nat, idl_nat, idr_nat)
}
```

pr_assoc, pr_idl, pr_idr are selectors of a category which return proof of associativity, left-identity and right-identity respectively. In the definition, {A|B} means the subtype of a type A satisfying B. We use subtyping to obtain the equality on natural transformations as definitional equality in the type system.

In the formalization T2, the equality of natural transformations is defined extensionally (`natext`).

```
{
  C:Cat(i,j)
  D:Cat(k,l)
  Functs = Functor(i,j,k,l) C D
  {
    F:Functs
    G:Functs
    NatTra = NatTrans C D F G
    natext = Πt1:NatTra.Πt2:NatTra.λa:Ob C.
              equal (Hom D (π1 F a) (π1 G a))
                    (π1 t1 a)
                    (π1 t2 a)
    Equive_nat_eq :(Equiv NatTra natext)
    Nat_setoid = (NatTra, natext, Equiv_nat_eq)
  }
  comp_nat : Comp_Type Functs Nat_setoid
  id_nat : ΠF:Functs.Nat_setoid F F
  assoc_nat : Assoc_law Functs Nat_setoid comp_nat
  idl_nat : Idl_law Functs Nat_setoid comp_nat id_nat
  idr_nat : Idr_law Functs Nat_setoid comp_nat id_nat
  Funct = (Functor(i,j,k,l) C D, Nat_setoid, comp_nat, id_nat,
             assoc_nat, idl_nat, idr_nat)
}
```

Chapter 4

Yoneda's lemma

Yoneda's lemma states that for each functor $\mathbf{K} : \mathbf{C} \rightarrow \mathbf{Set}$, $Nat(\mathbf{C}[r, \_], \mathbf{K})$ and $\mathbf{K}(r)$ are isomorphic. Here, $\mathbf{C}$ is a locally small category and $\mathbf{C}[r, \_]$ is hom-functor and category **Set** is the category of all sets. In the above, set in **GB** are used. Notion of set is twice here and **Set** is very important in category theory. We formalize proofs of the lemma and its application to discuss "notion of set" in each theory.

4.1 Definition of notions for Yoneda's lemma

The category **Set** is the category with the objects of all sets in bf BG, morphisms of all mappings between these sets with the usual composition of mappings.

In **BG**, the collection of all sets is not a set. But in formalization S, we can construct what corresponds to the collection as a set because S is based on **TG**. We construct the set V_0 of all **BG** sets in Chapter 5. In [5], for the set V of sets, the category Ens V is defined. Ens V is the category with the objects of all sets in V, morphisms of all mappings between these sets with the usual composition of mappings. Ens V was formalized as the followings.

```
:: Category Ens
reserve V for non empty set, A,B for Element of V;
definition let V;
  func Funcs(V) -> functional non empty set means
 it = union  Funcs(A,B): not contradiction ;
end;
reserve f for Element of Funcs(V);
definition let V;
  func Maps(V) -> non empty set means
 it =  [[A,B],f]: (B= ∅ implies A= ∅  & f is Function of A,B;
end;
reserve m,m1,m2 for Element of Maps V;
```

```
definition let V,m,m1,m2;
  func Dom V -> Function of Maps V,V means
 for m holds it.m = dom m;
  func Cod V -> Function of Maps V,V means
 for m holds it.m = cod m;
  func Comp V -> PartFunc of [:Maps V,Maps V:],Maps V means
   (for m2,m1 holds [m2,m1] ∈ dom it iff dom m2 = cod m1) &
   (for m2,m1 st dom m2 = cod m1  holds it.[m2,m1] = m2 • m1);
  func Id V -> Function of V,Maps V means
  for A holds it.A = id$ A;
end;
definition let V;
 func Ens(V) -> strict CatStr means
 it = CatStr≪ V,Maps V,Dom V,Cod V,Comp V,Id V ≫;
end;
theorem
   CatStr≪ V,Maps V,Dom V,Cod V,Comp V,Id V ≫ is Category;
```

In the above, dom and cod means selectors of elements(3-tuples) of Maps V and the notation " • " means the composition of elements of Maps V. We consider Ens(V_0) as the category Set.

In T1, as **Set**, we define the category SET of Type(0) as the followings.

```
map = λA:Type(0).λB:Type(0).A → B
comp_set =  λA:Type0.λB:Type0.λC:Type0.
            λf:B → C.λg:A → B.λx:A.f (g x)
assoc_set = λA:Type(0).λB:Type(0).λC:Type(0).λD:Type(0).
            λf:A → B.λg:B → C.λh:C → D.
               eqRef (A → D) λx:A.f (g (h x))
id_set = λA:Type(0).λx:A.x
idl_set =λA:Type(0).λB:Type(0).λg:B → A.eqRef (B → A) g
idr_set =λA:Type(0).λB:Type(0).λf:A → B.eqRef (A → B) f
SET = (Type(0), map, comp_set, id_set, assoc_set, idl_set, idr_set)
```

In T2, as **Set**, we define the category SET of Setoid(0) as the followings. (They are only declarations. See Appendix C for further particular.)

```
comp_set : Comp_Type Setoid0 Map_setoid
assoc_set : Assoc_law Setoid0 Map_setoid comp_set
id_set : Id_Type Setoid0 Map_setoid
```

```
idl_set : Idl_law Setoid0 Map_setoid comp_set id_set
idr_set : Idr_law Setoid0 Map_setoid comp_set id_set
SET = (Setoid(0), Map_setoid, comp_set, id_set, assoc_set, idl_set, idr_set)
```

Before hom-functors, we introduce opposite categories and contravariant functors. The opposite category $\mathbf{C}^op$ of a category C is defined as the below.

- $Ob_{\mathbf{C}^{op}} = Ob_{\mathbf{C}}$
- $Mor_{\mathbf{C}^{op}} = Mor_{\mathbf{C}}$
- $\mathbf{id}^op = \mathbf{id}$
- $\mathbf{dom}^op = \mathbf{cod}$
- $\mathbf{cod}^op = \mathbf{dom}$
- $f \circ^o pg = g \circ f$

Contravariant functors from $\mathbf{C}$ to $\mathbf{D}$ are functors from $\mathbf{C}^op$ to $\mathbf{D}$ and defined as the following. A **contravariant functor** $\mathbf{F} : \mathbf{C} \to \mathbf{D}$ is a pair of operations $\mathbf{F}_{ob} : Ob_{\mathbf{C}} \to Ob_{\mathbf{D}}, \mathbf{F}_{mor} : Mor_{\mathbf{C}} \to Mor_{\mathbf{D}}$ such that, for each $f : a \to b, g : b \to c$ in $\mathbf{C}$,

- $\mathbf{F}_{mor}(f) : \mathbf{F}_{ob}(b) \to \mathbf{F}_{ob}(a)$
- $\mathbf{F}_{mor}(g \circ f) = \mathbf{F}_{mor}(f) \circ \mathbf{F}_{mor}(g)$
- $\mathbf{F}_{mor}(id_a) = id_{\mathbf{F}_{ob(a)}}$.

For S, they are defined in [8] as the followings.

```
definition let X,Y,Z be non empty set; let f be PartFunc of [:X,Y:],Z;
 redefine func ~f -> PartFunc of [:Y,X:],Z;
end;
:: Opposite Category
definition let C;
  func C opp -> strict Category means
 it = CatStr ≪ the Objects of C, the Morphisms of C,
                  the Cod of C, the Dom of C,
                  ~(the Comp of C), the Id of C ≫;
end;
::  Contravariant Functors
definition let C,D;
 mode Contravariant_Functor of C,D
    -> Function of the Morphisms of C,the Morphisms of D means
```

```
    ( for c being Object of C ex d being Object of D st it.(id c) = id d )
  & ( for f being Morphism of C
       holds it.(id dom f) = id cod (it.f) & it.(id cod f) = id dom (it.f) )
  & ( for f,g being Morphism of C st dom g = cod f
       holds it.(g• f) = (it.f)• (it.g));
end;
```

For T1, we define contravariant functors in the below.

```
{
  C:Cat(i,j)
  D:Cat(k,l)
  {
    F1:Ob C → Ob D
    F2:Πa:Ob C.Πb:Ob C.Hom C a b → Hom D (F1 b) (F1 a)
    coComp_law = Πa:Ob C.Πb:Ob C.Πc:Ob C.
                 Πg:Hom C b c.Πf:Hom C a b.
                 ((F2 a c (o C a b c g f)) =
                  (o D (F1 c) (F1 b) (F1 a) (F2 a b f) (F2 b c g))
                   ∈ (Hom D (F1 c) (F1 a)))
    coId_law = (Πa:Ob C.(F2 a a (id C a)) = (id D (F1 a)) ∈ (Hom D (F1 a) (F1 a)))
  }
  ContraFunctor(i,j,k,l) = ΣF1:Ob C → Ob D.
                           ΣF2:Πa:Ob C.Πb:Ob C.Hom C a b → Hom D (F1 a) (F1 b)
                           (coComp_law F1 F2) × (coId_law F1 F2)
}
```

For T2, we define them as the followings.

```
{
  C:Cat(i,j)
  D:Cat(k,l)
  {
    F1:Ob C → Ob D
    F2:Πa:Ob C.Πb:Ob C.elem (Hom C a b) → elem (Hom D (F1 b) (F1 a))
    compC = comp (Ob C) (Hom C) (o D)
    compD = comp (Ob D) (Hom D) (o D)
    coComp_law = Πa:Ob C.Πb:Ob C.Πc:Ob C.
                 Πg:elem (Hom C b c).Πf:elem (Hom C a b).
```

```
                equal (Hom D (F1 c) (F1 a))
                  (F2 a c (compC a b c g f))
                  (compD (F1 c) (F1 b) (F1 a) (F2 a b f) (F2 b c g))
    coId_law = Πa:Ob C.equal (Hom D (F1 a) (F1 a)) (F2 a a (id C a)) (id D (F1 a))
  }
  ContraFunctor(i,j,k,l) = ΣF1:Ob C → Ob D.
                           ΣF2:Πa:Ob C.Πb:Ob C.elem (Hom C a b) → elem (Hom D (F1 a) (F1 b)).
                           (coComp_law F1 F2) × (coId_law F1 F2)
}
```

A category $\mathbf{C}$ is locally small when for all $a, b \in Ob_{\mathbf{C}}$, $\mathbf{C}[a,b]$ is a set in **GB**. If $\mathbf{C}$ is locally small and $a \in Ob_{\mathbf{C}}$, the (covariant) hom-functor $\mathbf{C}[a,\_] : \mathbf{C} \to \mathbf{Set}$ maps $b \in Ob_{\mathbf{C}}$ to $\mathbf{C}[a,b]$.
If $\mathbf{C}$ is locally small and $a \in Ob_{\mathbf{C}}$, the contravariant hom-functor $\mathbf{C}[\_,b] : \mathbf{C} \to \mathbf{Set}$ maps $a \in Ob_{\mathbf{C}}$ to $\mathbf{C}[a,b]$.

As the followings, we obtained the definition from [5] for S.

```
:: Hom-sets
reserve V for non empty set,
        C for Category,
        a,b for Object of C,
        f,g for Morphism of C;
definition let C,a,b;
   func Hom(a,b) -> set of Morphism of C means
 it =  f : dom(f)=a & cod(f)=b ;
end;
definition let C;
 func Hom(C) -> non empty set means
 it =  Hom(a,b): not contradiction ;
end;
:: hom-functors
definition let C,a,f;
 func hom(a,f) -> Function of Hom(a,dom f),Hom(a,cod f) means
 for g st g ∈ Hom(a,dom f) holds it.g = f • g;
 func hom(f,a) -> Function of Hom(cod f,a),Hom(dom f,a) means
 for g st g ∈ Hom(cod f,a) holds it.g = g • f;
end;
definition let C,a;
 func hom?-(a) -> Function of the Morphisms of C, Maps(Hom(C)) means
```

```
 for f holds it.f = [[Hom(a,dom f),Hom(a,cod f)],hom(a,f)];
 func hom-?(a) -> Function of the Morphisms of C, Maps(Hom(C)) means
 for f holds it.f = [[Hom(cod f,a),Hom(dom f,a)],hom(f,a)];
end;
theorem
   Hom(C) ⊆ V implies hom?-(a) is Functor of C,Ens(V);
theorem
   Hom(C) ⊆ V implies hom-?(a) is Contravariant_Functor of C,Ens(V);
definition let V,C,a;
 assume  Hom(C) ⊆ V;
 func hom?-(V,a) -> Functor of C,Ens(V) means
 it = hom?-(a);
 func hom-?(V,a) -> Contravariant_Functor of C,Ens(V) means
 it = hom-?(a);
end;
```

In it, assuming V ⊆ Hom(C), proved are that hom-functors are functors. In S, we consider only a category C such that $V_0 \subseteq$ Hom(C) as locally small category.

For T1, we define small category lsCat(i) as Cat(i,0) and hom-functor homr_ and contravariant hom-functor hom_r defined in the below.

```
lsCat(i) = Cat(i,0)
{
  C:lsCat(i)
  r:Ob C
  homr_ob = λa:Ob C.Hom C r a
  homr_mor = λa:Ob C.λb:Ob C.λf:Hom C a b.
              λg:Hom C r a.o C r a b f g
  pr_comp = λa:Ob C.λb:Ob C.λc:Ob C.
             λf:Hom C b c.λg:Hom C a b.
             eqSym (Hom C r c) (pr_assoc C r a b c f g h)
  pr_id = λa:Ob c.λf:Hom C r a.pr_idr a r f
  homr_ = (homr_ob, homr_mor, pr_comp, pr_id)

  hom_rob = λa:Ob C.Hom C a r
  hom_rmor = λa:Ob C.λb:Ob C.λf:Hom C a b.
              λh:Hom C b r.o C a b r h f
  pr_cocomp = λa:Ob C.λb:Ob C.λc:Ob C.
```

```
              λg:Hom C b c.λf:Hom C a b.
              λh:Hom C c r.
              (pr_assoc C a b c r h g f)
  pr_coid = λa:Ob C.λf:Hom C a r.
    eqSym (pr_idl C a r f)
  hom_r = (hom_rob, hom_rmor, pr_cocomp, pr_coid)
}
```

For T2, we also define small category lsCat(i) as Cat(i,0) and hom-functor homr_ and contravariant hom-functor hom_r defined in the below.

```
lsCat(i) = Cat(i,0)
{
  C:lsCat(i)
  r:Ob C
  oC = comp (Ob C) (Hom C) (o C)
  homr_ob = λa:Ob C.Hom C r a
  homr_mor = λa:Ob C.λb:Ob C.λf:elem (Hom C a b).
             (((λg:elem (Hom C r a).oC r a b f g),
               (λg1:elem (Hom C r a).λg2:elem (Hom C r a).
                λe:equal (π1 π2 C r a) g1 g2.
                π2 (π1 (o C r a b) f) g1 g2 e))
              :Map (Hom C r a) (Hom C r b))
  pr_comp = λa:Ob C.λb:Ob C.λc:Ob C.
            λf:elem (Hom C b c).λg:elem (Hom C a b).
            λh:elem (Hom C r a).
            pr_sym (Hom C r c) (oC r b c f (oC r a b g h))
                               (oC r a c (oC a b c f g) h)
                   (pr_assoc r a b c f g h)
  pr_id = λa:Ob C.λf:elem(Hom C r a).pr_idr a r f
  homr_ = (homr_ob, homr_mor, pr_comp, pr_id)

  hom_rob = λa:Ob C.Hom C a r
  hom_rmor = λa:Ob C.λb:Ob C.λf:elem (Hom C a b).
             (((λh:elem (Hom C b r).oC a b r h f),
               (λh1:elem (Hom C b r).λh2:elem (Hom C b r).
                λe:equal (π1 π2 C b r) h1 h2.
                π2 (o C a b r) h1 h2 e))
```

```
              :Map (Hom C b r) (Hom C a r))
  pr_cocomp = λa:Ob C.λb:Ob C.λc:Ob C.
              λg:elem (Hom C b c).λf:elem (Hom C a b).
              λh:elem (Hom C c r).
              (pr_assoc a b c r h g f)
  pr_coid = λa:Ob C.λf:elem (Hom C a r).pr_sym (Hom C a r) f (oC a a r f (id C a)) (pr_idl a r
  hom_r = (hom_rob, hom_rmor, pr_cocomp, pr_coid)
}
```

4.2 Formulation of proof of Yoneda's lemma

Yoneda's lemma states three propositions. For each functor $\mathbf{K} : \mathbf{C} \to \mathbf{Set}$, $Nat(\mathbf{C}[r,] , \mathbf{K})$ and $\mathbf{K}(r)$ are isomorphic with the map $Psi(r,K)$. And $Psi(r,K)$ is natural in r and K. As the common strategy, we prove that the yoneda diagram commutes, and then prove isomorphism and two kinds of natularity.

In S, we sketch the formalized proof of Yoneda lemma. The followings are formalized theorems.

```
reserve C for Category,
assume Hom(C) ⊆ V0;

theorem YonedaDiagram:
   for K being Functor of C,Ens(V0)
   for r,d being Objects of C
   for f being Morphism of r,d
   for phi being natural_transformation of hom?-(r),K
     holds (phi.d).f = (K.f).((phi.r).(id r));

definition let C,D be Category,
           let F,G be Functor of C,D;
Nat(F,G) means hom(F,G) of Functors(C,D);

definition let C be Category;
  let K be Functor of C,Ens(V0)
  let r be Objects of C
  func Psi(r,K) -> Function of Nat(hom?-(r),K),(Obj K).r means
    it.phi = (phi.r).(id r);
end;
```

```
theorem YonedaLemma:
  for K being Functor of C,Ens(V0)
  for r being Objects of C
    holds
      (for phi1,phi2 being natural_transformation of hom?-(r),K
         Psi(r,K).phi1 = Psi(r,K).phi2 implies phi1 = phi2)
    & (for x ∈ (Obj K).r
         ex phi be natural_transformation of hom?-(r),K such that
    Psi(r,K).phi = x);

definition let c,d,r be Object of C,f be Morphism of C;
  func hom?-(f) -> natural_transformation of hom?-(d),hom?-(r) means
  it.c = hom(f,c);
let phi be natural_transformation of hom?-(r),K;
  func -hom?-(f) -> Function of (natural_transformation of hom?-(r),K),
                                (natural_transformation of hom?-(d),K) means
  it.phi = phi∘hom?-(f);
end;

theorem PsiNatinr:
  for K being Functor of C,Ens(V0)
  for r,d being Objects of C
  for f being Morphism of r,d holds
    Psi(d,K)・-hom?-(f) = (K.f)・Psi(r,K);

theorem PsiNatinK:
  for K,H being Functor of C,Ens(V0)
  for r being Objects of C
  for mu being natural_transformation of K,H holds
    Psi(r,H)・?-(mu) = mu.r・Psi(r,K);
```

For T1, we prove the following isomorphism.

```
{
  C:lsCat(i)
  {
    r:Ob C
    K:Functor(i,0,1,0) C SET
```

```
    YonedaDiagram : Πphi:NatTrans C SET (homr_ C r) K.
                    Πd:Ob C.Πf:Hom r d.
                        ((π1 phi d) f
                       = (π1 π2 K r d f) ((π1 phi r) (id C r))
                       ∈ (π1 K r))
    {
      phi:NatTrans C SET (homr_ C r) K
      Psi = (π1 phi r) (id C r)
    }
    PsiInjective : Πphi1:NatTrans C SET (homr_ C r) K.
                   Πphi2:NatTrans C SET (homr_ C r) K.
                       (Psi phi1 = Psi phi2 ∈ (π1 K r))
                          → (Πd:Ob C.Πf:Hom C r d.(π1 phi1 d f
                          = π1 phi2 d f) ∈ (π1 K r))

    PsiSurjective : Πx:π1 K r.
                       ∃ phi1:NatTrans C SET (Hom_ C r) K.
                 (Psi phi1 = x ∈ (π1 K r))
    YonedaLemma = PsiInjective ∧ PsiSurjective
  }
}
```

For T2, we prove the following isomorphism.

```
{
  C:lsCat(i)
  {
    r:Ob C
    K:Functor(i,0,1,0) C SET
    YonedaDiagram : Πphi:NatTrans C SET (homr_ C r) K.
                    Πd:Ob C.Πf:elem(Hom r d).
                      equal (π1 K d)
                            (π1 (π1 phi d) f)
                            (π1 (π1 π2 K r d f) (π1 (π1 phi r) (id C r)))
    {
      phi:NatTrans C SET (homr_ C r) K
      Psi = π1 (π1 phi r) (id C r)
    }
```

```
    PsiInjective = Πphi1:NatTrans C SET (homr_ C r) K.
                   Πphi2:NatTrans C SET (homr_ C r) K.
                       equal (π1 K r) (Psi phi1) (Psi phi2)
                          → (Πd:Ob C.Πf:elem(Hom C r d).
                                equal ((π1 K d)
                                       (π1 (π1 phi1 d) f)
                                       (π1 (π1 phi2 d) f)
    PsiSurjective = Πx:elem(π1 K r). ∃ phi:NatTrans C SET (homr_ C r) K.
                       equal (π1 K r) (Psi phi) x
    YonedaLemma = PsiInjective ∧ PsiSurjective
  }
}
```

Chapter 5

Comparison of formalizations

5.1 Level of universe

For type theory T1 and T2, the types which have the same name and subscript is at the same level. In these formalization, the following judgements state.

- Setoid(i) : Type(i+1) ;The type of setoids.
- Cat(i,j) : Type(max(i,j)+1)
- lsCat(i) (= Cat(i,0)) : Type(i+1) ;The type of locally small categories
- Functor(i,j,k,l) C D : Type(max(i,j,k,l))
- Functor(i,0,j,0) C D : Type(max(i,j))
- Nattrans(i,j,k,l) C D F G : Type(max(i,j,l))
- Nattrans(i,0,j,0) C D F G : Type(i)
- Map_setoid(i) : Setoid(i)
- Funct(i,0,j,0) : Cat(i,0)

Map_setoid(i): the setoid of Maps from setoid(i) to setoid(i) is in setoid(i); that is, it is closed under exponent. Surprisingly, functor categories Funct(i,0,k,0) of locally small categories have the type Cat(i,0) and they are also locally small categories.

For formalization S, categorial categories are defined in [3] on Mizar system. It is notion at the higher level.

5.2 What is the category Set

The category **Set** plays an important role in category theory.

Required properties as the category **Set** are like the followings.

- Equality and judgement of membership exists on it.
- Closed under cartesian product and exponent.
- General set operations can be applied on it.
- ...

In type theory T1 and T2, we take the lowest universe (Type(0),Setoid(0) respectively) as the collection of all sets. In the formalizations in this paper, notion of set is used exactly twice. One is in the definition of small category and the other is in the definition of Set category. And same definitions are used twice.

They satisfy the above property and enough large to prove Yoneda's Lemma. In [4], naive set theory is introduced to ECC type theory based on the category **Set**. And free algebras such as monoids are defined based on it.

In the formalization S, we have to construct all **BG** sets in **TG** because there does not exists hierarchy.

To obtain it, we use the axiom Tarski9. Because of the axiom, **TG** has stronger power than **BG**.

```
theorem :: Tarski:9
   ex M st N ∈ M &
     (for X,Y holds X ∈ M & Y ⊆ X implies Y ∈ M) &
     (for X holds X ∈ M implies bool X ∈ M) &
     (for X holds X ⊆ M implies X≈M or X ∈ M);
```

The last line means that if a set included in M has the same cardinality as M it inhabits in M. Starting from empty set as N, we can get a enough large set M. We define V0 as a subset of the set M.

```
V0 = {x ∈ M||x|<|M|}
```

The set V0 is closed under cartesian product and function space. And Ens(V0) is regarded as the category **Set**.

5.3 Equality in type theories

Setoid in ECC can be considered as one of the quotient types. Because functions of equalities in these formalizations are extensionaly, The definition of Hom setoid in T2 are naive.

In T1, if the equality of mprphisms to define is different from the definitional equality, there are three measures.

- subtyping (valid cases are limited)

- quotient type (assuming the system support)
- Hom has a function to culculate the representatives.

For the definition of the functor category, subtyping can be used.

Chapter 6

Conclusion

We have completed formalization of Yoneda lemma on three systems(S, T1, T2). And level of types of basic notions in category theory are determined in type theory.

Formalization(S) based on set theory is unsuitable for formalizing category theory for lack of hierarchy because category theory is meta-theoretic. Formalization(T2) of category theory with setoids is powerful. But formalization(T1) with subtyping is enough for formalizing Yoneda's lemma and defining functor category.

Advanced these formalizations in this paper, more properties and notions can be defined as future work. Free algebras such as monoids and groups are defined on category theory, seen in [4]. In particular, Yoneda's lemma helps to formalize properties associated to topos.

As another advanced work, internal framework of naive set theory is constructed in type theory with the category **Set**.

Reference

[1] Altucher, J. A., and P. Panangaden, "A Mechanically Assisted Constructive Proof in Category Theory," in *CADE-10*, LNAI 449, pp. 500–513, Springer Verlag, 1990.

[2] Asperti, A., and G. Longo, *Categories, Types and Structures : An introduction to category theory for the working computer scientist.* Foundations of Computing Series, MIT Press, 1991.

[3] Bancerek, G., "Categorial Categories and Slice Categories," October 1994.

[4] Barthe, G., "Mathematical concepts in type theory." DRAFT.

[5] Bylińsky, C., "Category Ens," in *Formalized Mathematics*, vol. 2(4), pp. 527–533, Université Catholique de Louvain, 1991.

[6] Bylińsky, C., "Introduction to categories and functors," in *Formalized Mathematics*, vol. 1(2), pp. 409–420, Université Catholique de Louvain, 1990.

[7] Bylińsky, C., "Subcategories and Products of Categories," in *Formalized Mathematics*, vol. 1(4), pp. 725–732, Université Catholique de Louvain, 1990.

[8] Bylińsky, C., "Opposite Categories and Contravariant functors," in *Formalized Mathematics*, vol. 2(3), pp. 419–424, Université Catholique de Louvain, 1991.

[9] Bylińsky, C., "Products and Coproducts in Categories," in *Formalized Mathematics*, vol. 2(5), pp. 701–709, Université Catholique de Louvain, 1991.

[10] Bylińsky, C., "Cartesian Categories," in *Formalized Mathematics*, vol. 3(2), pp. 161–169, Université Catholique de Louvain, 1992.

[11] Constable, R., et al., *Implementing Mathematics with the Nuprl Proof Development System.* Englewood Cliffs, New Jersey 07632: Prentice-Hall, Inc., 1986.

[12] Coquand, T., and G. Huet, "The Calculus of Constructions," *Information and Computation*, vol. 76, no. 2/3, pp. 95–120, Feb./Mar. 1988.

[13] Curry, H., R. Feys, and W. Craig, *Combinatory Logic*, vol. 1. Amsterdam: Elsevier Science Publishers North-Holland, 1958.

[14] deBruijn, N., "A survey of the project AUTOMATH," in *Essays on Combinatory Logic, Lambda Calculus, and Formalism* (J. Seldin and J. Hindley, eds.), pp. 589–606, New York: Academic Press, 1980.

[15] Dowek, G., A. Felty, H. Herbelin, G. Huet, P. Paulin-Mohring, and B. Werner, *The Coq Proof Assistant User's Guide*. INRIA-Rochuencourt, CNRS-ENS Lyon, 1991.

[16] Gordon, M. J. C., "HOL: A Proof Generating System for Higher-Order Logic," in *VLSI Specification, Verification and Synthesis* (G. Birtwistle and P. A. Subrahmanyam, eds.), Kluwer International Series in Engineering and Computer Science, pp. 73–128, Kluwer Academic Publishers, 1988.

[17] Harper, R., and R. Pollack, "Universe Polymorphism," *Theoretical Computer Science*, vol. 89, 1991.

[18] Harper, R., F. Honsell, and G. Plotkin, "A Framework for Defining Logics," in *Proceedings, Symposium on Logic in Computer Science*, pp. 194–204, The Computer Society of the IEEE, 1987.

[19] Howard, W., "The formulas–as–types notion of construction," in *Essays on Combinatory Logic, Lambda Calculus, and Formalism* (J. Seldin and J. Hindley, eds.), pp. 479–490, New York: Academic Press, 1980.

[20] Huet, G., "Exisiting the Calculus of Constructions with Type:Type." manuscript, April 1987.

[21] Huet, G., and G. Plotkin, eds., *Proceedings of the 1st Workshop on Logical Frameworks*, ESPRIT BRA 3245, 1990.

[22] Huet, G., and G. Plotkin, eds., *Proceedings of the 2nd Workshop on Logical Frameworks*, ESPRIT BRA 3245, 1991.

[23] Lauchli, H., "An abstract notion of realizability for which intuitionistic predicate calculus is complete," in *Intuitionism and Proof Theory* (J. Myhill, A. Kino, and R. Vesley, eds.), Amsterdam: Elsevier Science Publishers North-Holland, 1970.

[24] Luo, Z., R. Pollack, and P. Taylor, "How to Use LEGO:a preliminary user's manual," LFCS Techincal Notes LFCS-TN-27, Dept. of Computer Science, University of Edinburgh, 1989.

[25] Luo, Z., "**ECC**, an Extended Calculus of Constructions," in *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, (Asilomar, California, U.S.A.), pp. 386–395, IEEE Computer Society Press, June 1989.

[26] Luo, Z., *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[27] Luo, Z., "Computation and reasoning: atpe theory for computer science." draft book, 1993.

[28] Martin-Löf, P., "An intuitionistic theory of types: predicative part," in *Logic Colloquium '73* (E. Rose and J. Shepherdson, eds.), pp. 73–118, Amsterdam: Elsevier Science Publishers North-Holland, 1973.

[29] Martin-Löf, P., "Constructive mathematics and computer programming," in *6th International Congress for Logic, Method, and Philosophy of Science (Hannover, Aug. 1979)*, pp. 153–175, Amsterdam: Elsevier Science Publishers North-Holland, 1982.

[30] Paulin-Mohring, C., "Inductive Definitions in the system Coq, Rules and Properties," in *Typed Lambda Calculi and Applications*, LNCS 664, Springer Verlag, 1993.

[31] Paulson, L., and T. Nipkow, *Isabelle Tutorial and User's Manual.* Computer Laboratory, University of Cambridge, 1991.

[32] R.Pollack., "The theory of LEGO." manuscript, 1988.

[33] Rudnicki, P., "An Overview of the MIZAR Project," tech. rep., Department of Computing Science, University of Alberta, June 1992.

[34] Trybulec, A., "Natural Transformations. Discrete Categories," in *Formalized Mathematics*, vol. 2(4), pp. 467–474, Université Catholique de Louvain, 1990.

Appendix A

Formalization S

```
::Category
struct CatStr
   ≪ Objects,Morphisms -> non empty set,
     Dom,Cod -> (Function of the Morphisms, the Objects),
     Comp -> (PartFunc of [:the Morphisms, the Morphisms :],the Morphisms),
     Id -> Function of the Objects, the Morphisms
   ≫;

definition
 mode Category-like->CatStr means
      (for f,g being Element of the Morphisms of it holds
         [g,f] ∈ dom(the Comp of it) iff (the Dom of it).g=(the Cod of it).f)
   &  (for f,g being Element of the Morphisms of it
         st (the Dom of it).g=(the Cod of it).f holds
         (the Dom of it).((the Comp of it).[g,f]) = (the Dom of it).f &
         (the Cod of it).((the Comp of it).[g,f]) = (the Cod of it).g)
    & (for f,g,h being Element of the Morphisms of it
         st (the Dom of it).h = (the Cod of it).g &
            (the Dom of it).g = (the Cod of it).f
        holds (the Comp of it).[h,(the Comp of it).[g,f]]
            = (the Comp of it).[(the Comp of it).[h,g],f] )
    & (for b being Element of the Objects of it holds
         (the Dom of it).((the Id of it).b) = b &
         (the Cod of it).((the Id of it).b) = b &
         (for f being Element of the Morphisms of it st (the Cod of it).f = b
           holds (the Comp of it).[(the Id of it).b,f] = f ) &
         (for g being Element of the Morphisms of it st (the Dom of it).g = b
```

```
          holds (the Comp of it).[g,(the Id of it).b] = g ) );
end;

definition
 mode Category is Category-like CatStr;
end;
definition
 cluster strict Category;
end;



::Functor
reserve C,D for Category;
definition let C,D;
 mode Functor of C,D -> Function of the Morphisms of C,the Morphisms of D
   means
    ( for c being Element of the Objects of C
       ex d being Element of the Objects of D
        st it.((the Id of C).c) = (the Id of D).d )
    & ( for f being Element of the Morphisms of C holds
         it.((the Id of C).((the Dom of C).f)) =
            (the Id of D).((the Dom of D).(it.f)) &
         it.((the Id of C).((the Cod of C).f)) =
            (the Id of D).((the Cod of D).(it.f)) )
    & ( for f,g being Element of the Morphisms of C
           st [g,f] ∈ dom(the Comp of C)
         holds it.((the Comp of C).[g,f]) = (the Comp of D).[it.g,it.f] );
end;



:: Object Function of a Functor
definition let C,D;
 let F be Function of the Morphisms of C,the Morphisms of D;
 assume
 for c being Element of the Objects of C
    ex d being Element of the Objects of D
     st F.((the Id of C).c) = (the Id of D).d;
```

```
 func Obj(F) -> Function of the Objects of C,the Objects of D means
  for c being Element of the Objects of C
    for d being Element of the Objects of D
     st F.((the Id of C).c) = (the Id of D).d holds it.c = d;
end;



reserve a,b for Object of C;
definition let C,a,b;
  assume  Hom(a,b)<> ∅
  mode Morphism of a,b -> Morphism of C means
  it ∈ Hom(a,b);
end;



::Transformations
reserve A,B for Category,
        F,F1,F2 for Functor of A,B;

definition let A,B,F1,F2;
 pred F1 is_transformable_to F2 means
 for a being Object of A holds Hom(F1.a,F2.a) <> ∅
end;

definition let A,B,F1,F2;
 assume  F1 is_transformable_to F2;
 mode transformation of F1,F2 ->
         Function of the Objects of A, the Morphisms of B means
 for a being Object of A holds it.a is Morphism of F1.a,F2.a;
end;

definition let A,B;
 func id F ->transformation of F,F means
 for a being Object of A holds it.a = id (F.a);
end;
```

```
definition let A,B,F1,F2;
 assume  F1 is_transformable_to F2;
 let t be transformation of F1,F2; let a be Object of A;
 func t.a -> Morphism of F1.a, F2.a means
 it = t.a;
end;

definition let A,B,F,F1,F2;
 assume that
 F is_transformable_to F1 and
 F1 is_transformable_to F2;
 let t1 be transformation of F,F1;
 let t2 be transformation of F1,F2;
 func t2∘t1 -> transformation of F,F2 means
 for a being Object of A holds it.a = (t2.a)• (t1.a);
end;


:: Natural transformations
definition let A,B,F1,F2;
 pred F1 is_naturally_transformable_to F2 means
 F1 is_transformable_to F2 &
 ex t being transformation of F1,F2 st
  for a,b being Object of A st Hom(a,b) <> ∅
   for f being Morphism of a,b holds t.b• F1.f = F2.f• t.a;
end;

definition let A,B,F1,F2;
 assume F1 is_naturally_transformable_to F2;
 mode natural_transformation of F1,F2 -> transformation of F1,F2 means
  for a,b being Object of A st Hom(a,b) <> ∅ for f being Morphism of a,b holds it.b•
F1.f = F2.f• it.a;
end;

definition let A,B,F;
 redefine func id F -> natural_transformation of F,F;
```

```
end;

definition let A,B,F,F1,F2 such that
 F is_naturally_transformable_to F1 and
 F1 is_naturally_transformable_to F2;
 let t1 be natural_transformation of F,F1;
 let t2 be natural_transformation of F1,F2;
 func t2∘t1 -> natural_transformation of F,F2 means
 it = t2∘t1;
end;



:: Functor category
reserve A,B for Category;

definition let A,B;
 mode NatTrans-DOMAIN of A,B -> non empty set means
 for x being Any holds x ∈ t implies
  ex F1,F2 being Functor of A,B, t being natural_transformation of F1,F2
   st x = [[F1,F2],t] & F1 is_naturally_transformable_to F2;
end;

definition let A,B;
 func NatTrans(A,B) -> NatTrans-DOMAIN of A,B means
 for x being Any holds x ∈ it iff
  ex F1,F2 being Functor of A,B, t being natural_transformation of F1,F2
   st x = [[F1,F2],t] & F1 is_naturally_transformable_to F2;
end;

definition let A,B;
 func Functors(A,B) -> strict Category means
 the Objects of it = Funct(A,B) &
   the Morphisms of it = NatTrans(A,B) &
   (for f being Morphism of it holds dom f = f‘1‘1 & cod f = f‘1‘2) &
   (for f,g being Morphism of it st dom g = cod f
     holds [g,f] ∈ dom the Comp of it) &
   (for f,g being Morphism of it st [g,f] ∈ dom (the Comp of it)
```

```
     ex F,F1,F2,t,t1 st f = [[F,F1],t] & g = [[F1,F2],t1] &
      (the Comp of it).[g,f] = [[F,F2],t1∘t]) &
   for a being Object of it, F st F = a holds id a = [[F,F],id F];
end;



:: Category Ens
reserve V for non empty set, A,B for Element of V;
definition let V;
 func Funcs(V) -> functional non empty set means
 it = union  Funcs(A,B): not contradiction ;
end;
reserve f for Element of Funcs(V);
definition let V;
 func Maps(V) -> non empty set means
 it =  [[A,B],f]: (B= ∅ implies A= ∅  & f is Function of A,B;
end;
reserve m,m1,m2 for Element of Maps V;
definition let V,m;
 func graph(m) -> Function means
 it = m‘2;
 func dom m -> Element of V means
 it = m‘1‘1;
 func cod m -> Element of V means
 it = m‘1‘2;
end;
definition let V,m1,m2;
  assume  cod m1 = dom m2;
   func m2 • m1 -> Element of Maps V means
 it = [[dom m1,cod m2],graph(m2) • graph(m1)];
end;
definition let V;
  func Dom V -> Function of Maps V,V means
 for m holds it.m = dom m;
  func Cod V -> Function of Maps V,V means
 for m holds it.m = cod m;
  func Comp V -> PartFunc of [:Maps V,Maps V:],Maps V means
```

```
   (for m2,m1 holds [m2,m1] ∈ dom it iff dom m2 = cod m1) &
   (for m2,m1 st dom m2 = cod m1  holds it.[m2,m1] = m2 • m1);
  func Id V -> Function of V,Maps V means
  for A holds it.A = id$ A;
end;
definition let V;
 func Ens(V) -> strict CatStr means
 it = CatStr≪ V,Maps V,Dom V,Cod V,Comp V,Id V ≫;
end;
theorem
   CatStr≪ V,Maps V,Dom V,Cod V,Comp V,Id V ≫ is Category;
definition let V;
 cluster Ens(V) -> Category-like;
end;



definition let X,Y,Z be non empty set; let f be PartFunc of [:X,Y:],Z;
 redefine func ~f -> PartFunc of [:Y,X:],Z;
end;
:: Opposite Category
definition let C;
  func C opp -> strict Category means
:: OPPCAT_1: def 1
 it = CatStr ≪the Objects of C, the Morphisms of C,
                   the Cod of C, the Dom of C,
                   ~(the Comp of C), the Id of C ≫;
end;
::  Contravariant Functors
definition let C,D;
 mode Contravariant_Functor of C,D
    -> Function of the Morphisms of C,the Morphisms of D means
    ( for c being Object of C ex d being Object of D st it.(id c) = id d )
  & ( for f being Morphism of C
       holds it.(id dom f) = id cod (it.f) & it.(id cod f) = id dom (it.f) )
  & ( for f,g being Morphism of C st dom g = cod f
       holds it.(g• f) = (it.f)• (it.g));
end;
```

```
:: Hom-sets
reserve V for non empty set,
        C for Category,
        a,b for Object of C,
        f,g for Morphism of C;
definition let C,a,b;
   func Hom(a,b) -> set of Morphism of C means
 it =  f : dom(f)=a & cod(f)=b ;
end;
definition let C;
 func Hom(C) -> non empty set means
 it =  Hom(a,b): not contradiction ;
end;
:: hom-functors
definition let C,a,f;
 func hom(a,f) -> Function of Hom(a,dom f),Hom(a,cod f) means
 for g st g ∈ Hom(a,dom f) holds it.g = f • g;
 func hom(f,a) -> Function of Hom(cod f,a),Hom(dom f,a) means
 for g st g ∈ Hom(cod f,a) holds it.g = g • f;
end;
definition let C,a;
 func hom?-(a) -> Function of the Morphisms of C, Maps(Hom(C)) means
 for f holds it.f = [[Hom(a,dom f),Hom(a,cod f)],hom(a,f)];
 func hom-?(a) -> Function of the Morphisms of C, Maps(Hom(C)) means
 for f holds it.f = [[Hom(cod f,a),Hom(dom f,a)],hom(f,a)];
end;
theorem
   Hom(C) ⊆ V implies hom?-(a) is Functor of C,Ens(V);
theorem
   Hom(C) ⊆ V implies hom-?(a) is Contravariant_Functor of C,Ens(V);
definition let V,C,a;
 assume  Hom(C) ⊆ V;
 func hom?-(V,a) -> Functor of C,Ens(V) means
 it = hom?-(a);
 func hom-?(V,a) -> Contravariant_Functor of C,Ens(V) means
```

```
 it = hom-?(a);
end;



reserve C for Category,
assume Hom(C) ⊆ V0;

theorem YonedaDiagram:
   for K being Functor of C,Ens(V0)
   for r,d being Objects of C
   for f being Morphism of r,d
   for phi being natural_transformation of hom?-(r),K
     holds (phi.d).f = (K.f).((phi.r).(id r))
  proof let K be Functor of C,Ens(V0);
        let r,d be Objects of C;
let f be Morphism of r,d;
        let phi be natural_transformation of hom?-(r),K;
    (phi.d).f = (phi.d).(f• (id r)) by CAT_1:DEF8
             .= (phi.d).(hom(r,f).(id r)) by ENS_1:DEF20
             .= ((phi.d)• hom(r,f)).(id r) by FUNCT_1:23,ENS_1:DEF3,ENS_1:DEF7,ENS_1:DEF14
     .= ((phi.d)• (hom?-(r).f)).(id r) by ENS_1:DEF7,DEF12,DEF14,DEF22
             .= ((K.f)• (phi.r)).(id r) by NATTRA_1:DEF7
             .= (K.f).((phi.r).(id r)) by FUNCT_1:22,ENS_1:DEF3,ENS_1:DEF7,ENS_1:DEF14
end;

definition let C,D be Category,
           let F,G be Functor of C,D;
Nat(F,G) means hom(F,G) of Functors(C,D);

definition let C be Category;
  let K be Functor of C,Ens(V0)
  let r be Objects of C
  func Psi(r,K) -> Function of Nat(hom?-(r),K),(Obj K).r means
    it.phi = (phi.r).(id r);
end;

theorem YonedaLemma:
```

```
  for K being Functor of C,Ens(V0)
  for r being Objects of C
    holds
      (for phi1,phi2 being natural_transformation of hom?-(r),K
         Psi(r,K).phi1 = Psi(r,K).phi2 implies phi1 = phi2)
    & (for x ∈ (Obj K).r
         ex phi be natural_transformation of hom?-(r),K such that
    Psi(r,K).phi = x)
  proof let K be Functor of C,Ens(V0),r be Objects of C;
    thus for phi1,phi2 being natural_transformation of hom?-(r),K
          Psi(r,K).phi1 = Psi(r,K).phi2 implies phi1 = phi2
      proof let phi1,phi2 be natural_transformation of hom?-(r),K
        let d be Objects of C
        assume
A1:       si(r,K).phi1 = Psi(r,K).phi2;
        then
          (phi1.d).f = (K.f).((phi1.r).(id r)) by YonedaDiaram
                     .= (K.f).(Psi(r,K).phi1)   by definition of Psi
                     .= (K.f).(Psi(r,K).phi2)   by A1
                     .= (K.f).((phi2.r).(id r)) by definition of Psi
                     .= (phi2.d).f              by YonedaDiagram
      end;
      proof assume x ∈ (Obj K).r
        consider phi be transformation of hom?-(r),K such that
A2:       for d being Object of C, f being Morphism of r,d
            (phi.d).f = (K.f).x
          thus phi is natural_transformation of hom?-(r),K
    proof let a,b be Object of C;
            let f be Morphism of a,b,
                g be Mprphism of r,a;
              ((K.f)・(phi.a)).g = (K.f).((phi.a).g)          by FUNCT_1:22
                                .= (K.f).((K.g).x)           by A2
                                .= ((K.f)・(K.g)).x          by FUNCT_1:23
                                .= (K.(f・g)).x              by CAT_1:DEF18
                                .= (phi.b).(f・g)            by A2
                                .= (phi.b).(hom(a,f).g)      by ENS_1:DEF20
                                .= ((phi.b)・(hom(a,f)).g    by FUNCT_1:23
```

```
                                  .= ((phi.b)• (hom?-(a).f)).g  by ENS_1:DEF22
    end;
  thus Psi.phi = x
           proof
              Psi.phi = (phi.r).(id r)      by definition of Psi
                      = (K.(id r)).x        by A2
                      = (id ((Obj K).r)).x  by CAT_1:DEF18
                      = x                   by ENS_1:DEF6,DEF14
           end;
      end;
  end;

definition let c,d,r be Object of C,f be Morphism of C;
  func hom?-(f) -> natural_transformation of hom?-(d),hom?-(r) means
  it.c = hom(f,c);
let phi be natural_transformation of hom?-(r),K;
  func -hom?-(f) -> Function of (natural_transformation of hom?-(r),K),
                                (natural_transformation of hom?-(d),K) means
  it.phi = phi∘hom?-(f);
end;

theorem PsiNatinr:
  for K being Functor of C,Ens(V0)
  for r,d being Objects of C
  for f being Morphism of r,d holds
    Psi(d,K)• -hom?-(f) = (K.f)• Psi(r,K)
  proof
    let phi be natural_transformation of hom?-(r),K ;
    (Psi(d,K)• -hom?-(f)).phi = Psi(d,K).(-hom?-(f).phi)
                              .= Psi(d,K).(phi∘hom?-(f))
                              .= (phi∘hom?-(f)).d).(id d)
                              .= ((phi.d)• hom(f,d)).(id d)
                              .= (phi.d).(hom(f,d).(id d))
                              .= (phi.d).((id d)• f)
                              .= (phi.d).f
                              .= (K.f).((phi.r).(id r))  by YonedaDiagram
                              .= (K.f).(Psi(r,K).phi)
```

```
                              .= ((K.f)• (Psi(r,K)).phi

definition let K,H be Functor of C,Ens(V0),
               mu be natural_transformation of K,H,
               tau be natural_transformation of hom?-(r),K;
  func ?-(mu) -> Function of Nat(hom?-(r),K),Nat(hom?-(r),H) means
  it.tau = mu∘tau;
end;

theorem PsiNatinK:
  for K,H being Functor of C,Ens(V0)
  for r being Objects of C
  for mu being natural_transformation of K,H holds
    Psi(r,H)•?-(mu) = mu.r• Psi(r,K)
  proof
    let K,H be Functor of C,Ens(V0);
    let r be Objects of C;
    let mu being natural_transformation of K,H;
    let phi be natural_transformation of hom?-(r),K;
      (Psi(r,H)•?-(mu)).phi = Psi(r,H).(?-(mu).phi)
                            .= Psi(r.H).(mu∘phi)
                            .= ((mu∘phi).r).(id r)
                            .= (mu.r• phi.r).(id r)
                            .= (mu.r).((phi.r).(id r))
                            .= (mu.r).(Psi(r,K).phi)
                            .= ((mu.r)• Psi(r,K)).phi
  end;
```

Appendix B

Formalization T1

```
;;;Definition of the type Cat of categories
{
  Ob:Type(i)
  H:Ob → Ob → Type(j)
  Comp_Type = Πa:Ob.Πb:Ob.Πc:Ob.(H b c) → (H a b) → (H a c)
  Id_Type = Πa:Ob.H a a
  o:Comp_Type
  Assoc_law = Πa:Ob.Πb:Ob.Πc:Ob.Πd:Ob.
              Πf:H c d.Πg:H b c.Πh:H a b.
              o a c d f (o a b c g h) = o a b d (o b c d f g) h
                                       ∈ H a d
  id:Id_Type
  Idl_law = Πa:Ob.Πb:Ob.Πf:H a b.f = o a a b f (id a) ∈ H a b
  Idr_law = Πa:Ob.Πb:Ob.Πg:H b a.o b a a (id a) g = g ∈ H b a
}
Cat(i,j) = ΣOb:Type(i).ΣH:Ob → Ob → Type(j).
           Σo:Comp_Type Ob H.Σid:Id_Type Ob H.
           (Assoc_law Ob H o) × (Idl_law Ob H o id) × (Idr_law Ob H o id)

;;;Selectors for Cat
{
  C:Cat(i,j)
  Ob = π1 C
  Hom = π1 π2 C
  o = π1 π2 π2 C
  id = π1 π2 π2 π2 C
  pr_assoc = π1 π2 π2 π2 π2 C
```

```
  pr_idl = π1 π2 π2 π2 π2 π2 C
  pr_idr = π2 π2 π2 π2 π2 π2 C
}

;;;Definition of the type Functor of functors
{
  C:Cat(i,j)
  D:Cat(k,l)
  {
    Fob:Ob C → Ob D
    Fmor:Πa:Ob C.Πb:Ob C.Hom C a b → Hom D (Fob a) (Fob b)
    Comp_law = Πa:Ob C.Πb:Ob C.Πc:Ob C.
               Πf:Hom C b c.Πg:Hom C a b.
               Fmor a c (o C a b c f g) = o D (Fob a) (Fob b) (Fob c)
                                             (Fmor b c f) (Fmor a b g)
                                          ∈ Hom D (Fob a) (Fob c)
    Id_law = Πa:Ob C.Fmor a a (id C a) = id D (Fob a) ∈ Hom D (Fob a) (Fob a)
  }
  Functor(i,j,k,l) = ΣFob:Ob C → Ob D.
                     ΣFmor:Πa:Ob C.Πb:Ob C.Hom C a b → Hom D (Fob a) (Fob b).
                     (Comp_law Fob Fmor) × (Id_law Fob Fmor)
}

;;;Definition the type ContraFunctor of contravariant functors
{
  C:Cat(i,j)
  D:Cat(k,l)
  {
    Fob:Ob C → Ob D
    Fmor:Πa:Ob C.Πb:Ob C.Hom C a b → Hom D (Fob b) (Fob a)
    coComp_law = Πa:Ob C.Πb:Ob C.Πc:Ob C.
                 Πg:Hom C b c.Πf:Hom C a b.
                 Fmor a c (o C a b c g f) = o D (Fob c) (Fob b) (Fob a)
                                               (Fmor a b f) (Fmor b c g)
                                            ∈ Hom D (Fob c) (Fob a)
    coId_law = Πa:Ob C.Fmor a a (id C a) = id D (Fob a) ∈ Hom D (Fob a) (Fob a)
  }
```

```
  ContraFunctor(i,j,k,l) = ΣFob:Ob C → Ob D.
                           ΣFmor:Πa:Ob C.Πb:Ob C.Hom C a b → Hom D (Fob b) (Fob a).
                           (coComp_law Fob Fmor) × (coId_law Fob Fmor)
}

;;;Definition of the type NatTrans of natural transformations
{
  C:Cat(i,j)
  D:Cat(k,l)
  F:Functor C D
  G:Functor C D
  NatTrans(i,j,k,l) = Σtau:Πa:Ob C.Hom D (π1 F a) (π1 G a).
                      Πa:Ob C.Πb:Ob C.Πf:Hom C a b.
                         o D (π1 F a) (π1 G a) (π1 G b) (π1 π2 G a b f) (tau a)
                      = o D (π1 F a) (π1 F b) (π1 G b) (tau b) (π1 π2 F a b f)
                      ∈ Hom D (π1 F a) (π1 G b)
}

;;;category SET of all sets
Map = λ A:Type(0).λB:Type(0).A → B
comp_set = λA:Type(0).λB:Type(0).λC:Type(0).λf:B → C.λg:A → B.λx:A.f (g x)
Map_comp = λA|Type(0).λB|Type(0).λC|Type(0).comp_set A B C
{
  A:Type(0)
  B:Type(0)
  C:Type(0)
  D:Type(0)
  f:Map C D
  g:Map B C
  h:Map A B
  assoc_set = eqRef ∈ (λx:A.f (g (h x))) = (λx:A.f (g (h x)))
}
assoc_set ∈ Assoc_law Type(0) Map comp_set
id_set = λA:Type(0).λx:A.x
{
  A:Type(0)
  B:Type(0)
```

```
  f:A → B
  idl_set = eqExt f (λx:A.f x) (λx:A.(eqRef:f x = f x ∈ B))
}
idl_set ∈ Idl_law Type(0) Map comp_set id_set
{
  A:Type(0)
  B:Type(0)
  g:B → A
  idr_set = eqExt (λx:B.g x) g (λx:B.(eqRef:g x = g x ∈ A))
}
idr_set ∈ Idr_law Type(0) Map comp_set id_set
SET = (Type(0), Map, comp_set, id_set, assoc_set, idl_set, idr_set) ∈ Cat(1,0)

;;;functor category
{
  C:Cat(i,j)
  D:Cat(k,l)
  Functs = Functor C D
  NatTr = {π1 (NatTrans C D F G)|π2 (NatTrans C D F G)}
  {
    F1:Functs
    F2:Functs
    F3:Functs
    t1:NatTr F2 F3
    t2:NatTr F1 F2
    t1t2 = λa:Ob C.o D (π1 F1 a) (π1 F2 a) (π1 F3 a) (π1 t1 a) (π1 t2 a)
    {
      a:Ob C
      b:Ob C
      f:Hom C a b
      ;F3(f) • (t1(a)) • t2(a))=(t1(b) • t2(b)) • F1(f)
      {
        ;F3(f) • t1(a)=t1(b) • F2(f)
        t1nat = π2 t1 a b f
        ;F2(f) • t2(a)=t2(b) • F1(f)
        t2nat = π2 t2 a b f
        ;F3(f) • (t1(a)) • t2(a))=(F3(f) • t1(a)) • t2(a)
```

```
      _0 = pr_assoc D (π1 F1 a) (π1 F2 a) (π1 F3 a) (π1 F3 b)
                      ((π1 π2 F3 a b) f) (π1 t1 a) (π1 t2 a)
      ;t1(b) • (F2(f) • t2(a))=(t1(b) • F2(f)) • t2(a)
      _1 = pr_assoc D (π1 F1 a) (π1 F2 a) (π1 F2 b) (π1 F3 b)
                      (π1 t1 b) ((π1 π2 F2 a b) f) (π1 t2 a)
      ;t1(b) • (t2(b) • F1(F))=(t1(b) • t2(b)) • F1(F)
      _2 = pr_assoc D (π1 F1 a) (π1 F1 b) (π1 F2 b) (π1 F3 b)
                      (π1 t1 b) (π1 t2 b) ((π1 π2 F1 a b) f)
      ;t1(b) • F2(f) • t2(a)=t1(b) • t2(b) • F1(f)
      t1xt2Isnat = eqEq t2nat (o1 D (π1 F1 a) (π1 F2 b) (π1 F3 b) (π1 t1 b))
      ;F3(f) • t1(a) • t2(a)=t1(b) • F2(f) • t2(a)
      t1Isnatxt2 = eqEq t1nat
                       (λh:Hom D (π1 F2 a) (π1 F3 b).
                         (o1 D (π1 F1 a) (π1 F2 a) (π1 F3 b) h (π1 t2 a)))
      t1t2isNatural = eqTrans _0
                              (eqTrans t1Isnatxt2
                                       (eqTrans (eqSym _1)
                                                (eqTrans t1xt2Isnat _2)))
    }
  }
  comp_functs = (t1t2, t1t2isNatural) ∈ NatTr F1 F3
}
{
  F:Functs
  iota = λa:Ob C.(id D (π1 F a))
  {
    a:Ob C
    b:Ob C
    f:Hom C a b
    iotaisNatural = eqSym
                      (eqTrans (pr_idr D (π1 F b) (π1 F a) ((π1 π2 F a b) f))
                               (pr_idl D (π1 F a) (π1 F b) ((π1 π2 F a b) f)))
  }
  id_functs = (iota, iotaisNatural) ∈ NatTr F F
}
id_functs ∈ Id_Type Functs NatTr
{
```

```
    F1:Functs
    F2:Functs
    F3:Functs
    F4:Functs
    t1:NatTr F3 F4
    t2:NatTr F2 F3
    t3:NatTr F1 F2
    _ = eqExt (π1 (comp_functs F1 F3 F4 t1 (comp_functs F1 F2 F3 t2 t3)))
              (π1 (comp_functs F1 F2 F4 (comp_functs F2 F3 F4 t1 t2) t3))
              (λa:Ob C.pr_assoc D (π1 F1 a) (π1 F2 a) (π1 F3 a) (π1 F4 a)
                                  (π1 t1 a) (π1 t2 a) (π1 t3 a))
    assoc_functs = _
  }
  assoc_functs ∈ Assoc_law Functs NatTr comp_functs
  {
    F:Functs
    G:Functs
    t:NatTr F G
    {
      _ = eqExt (π1 t) (π1 (comp_functs F F G t (id_functs F))) (λa:Ob C.pr_idl (π1 F a) (π1 G
      idl_functs = _ ∈ t = comp_functs F F G t (id_functs F)
    }
  }
  idl_functs ∈ Idl_law Functs NatTr comp_functs id_functs
  {
    F:Functs
    G:Functs
    t:NatTr G F
    {
      _ = eqExt (π1 (comp_functs G F F  (id_functs F) t)) (π1 t)
                (λa:Ob C.pr_idr (π1 F a) (π1 G a) (π1 t a))
      idr_functs = _
    }
  }
  idr_functs ∈ Idr_law Functs NatTr comp_functs id_functs
  Funct = (Functs, NatTr, comp_functs, id_functs, assoc_functs, idl_functs, idr_functs)
}
```

```
{
  C:Cat(i,0)
  r:Ob C
  homr_ob = λa:Ob C.Hom C r a
  homr_mor = λa:Ob C.λb:Ob C.λf:Hom C a b.λg:Hom C r a.o C r a b f g
  {
    a:Ob C
    b:Ob C
    c:Ob C
    f:Hom C b c
    g:Hom C a b
    pr_comp = eqExt (homr_mor a c (o C a b c f g))
                    (o SET (homr_ob a) (homr_ob b) (homr_ob c)
                           (homr_mor b c f) (homr_mor a b g))
                    (λh:Hom C r a.eqSym (pr_assoc C r a b c f g h))
  }
  pr_id = λa:Ob C.eqExt (homr_mor a a (id C a))
                        (id SET (homr_ob a))
                        (λf:Hom C r a.pr_idr C a r f)
  ;covariant hom-functor
  homr_ = (homr_ob, homr_mor, pr_comp, pr_id) ∈ Functor C SET

  hom_rob = λa:Ob C.Hom C a r
  hom_rmor = λa:Ob C.λb:Ob C.λf:Hom C a b.λh:Hom C b r.o C a b r h f
  {
    a:Ob C
    b:Ob C
    c:Ob C
    g:Hom C b c
    f:Hom C a b
    pr_cocomp = eqExt (hom_rmor a c (o C a b c g f))
                      (o SET (hom_rob c) (hom_rob b) (hom_rob a)
                             (hom_rmor a b f) (hom_rmor b c g))
                      (λh:Hom C c r.pr_assoc C a b c r h g f)
  }
  pr_coid = λa:Ob C.eqExt (hom_rmor a a (id C a))
```

```
                          (id SET (hom_rob a))
                          (λf:Hom C a r.eqSym (pr_idl C a r f))
  ;contravariant hom-functor
  hom_r = (hom_rob, hom_rmor, pr_cocomp, pr_coid) ∈ ContraFunctor C SET
}

;;;Yoneda
{
  C:Cat(i,0)
  {
    r:Ob C
    K:Functor C SET
    {
      phi:NatTrans C SET (homr_ C r) K
      d:Ob C
      f:Hom C r d
      _1 = eqEq (pr_idl C r d f) (π1 phi d)
      _2 = eqSym (eqEq (π2 phi r d f) (λg:(Hom C r r)→(π1 K d).g (id C r)))
      YonedaDiagram = eqTrans _1 _2
    }
    {
      phi:NatTrans C SET (homr_ C r) K
      Psi = (π1 phi r) (id C r)
    }
    {
      phi1:NatTrans C SET (homr_ C r) K
      phi2:NatTrans C SET (homr_ C r) K
      assume:Psi phi1 = Psi phi2 ∈ π1 K r
      {
        d:Ob C
        f:Hom C r d
        _1 = YonedaDiagram phi1 d f
        _2 = YonedaDiagram phi2 d f
        _3 = eqEq assume (π1 π2 K r d f)
        PsiInjective = eqTrans _1 (eqTrans _3 (eqSym _2))
      }
    }
```

```
  {
    x:π1 K r
    phi = λd:Ob C.λf:Hom C r d.(π1 π2 K r d f) x
    {
      a:Ob C
      b:Ob C
      f:Hom C a b
      phiNatural = eqExt (o1 SET (Hom C r a) (π1 K a) (π1 K b)
                                 (π1 π2 K a b f) (phi a))
                         (o1 SET (Hom C r a) (Hom C r b) (π1 K b)
                                 (phi b) (homr_mor C r a b f))
                         (λg:Hom C r a.eqSym (eqEq (π1 π2 π2 K r a b f g)
                                                   (λh:(π1 K r → π1 K b).h x)))
    }
    phiNat = (phi, phiNatural) ∈ NatTrans C SET (homr_ C r) K
    _0 = eqEq (π2 π2 π2 K r) (λidKr:(π1 K r) → (π1 K r).idKr x)
    PsiSurjective = exI phiNat _0
  }
  IsoType = (Πphi1:NatTrans C SET (homr_ C r) K.
             Πphi2:NatTrans C SET (homr_ C r) K.
             (Psi phi1 = Psi phi2 ∈ π1 K r) →
             Πd:Ob C.Πf:Hom C r d.(π1 phi1 d) f = (π1 phi2 d) f) ∈ π1 K d
          ∧ (Πx:π1 K r. ∃ phiNat:NatTrans C SET (homr_ C r) K.
                          (Psi phiNat) = x ∈ π1 K r)
  YonedaLemma = andI PsiInjective PsiSurjective ∈ IsoType
}
YonedaLemma ∈ Πr:Ob C.ΠK:Functor C SET.IsoType r K
{
  r:Ob C
  d:Ob C
  K:Functor C SET
  f:Hom C r d
  ;natural transformantion C[f,_]:C[d,_] → C[r,_]
  Cf_ = λc:Ob C.hom_rmor C c r d f
  {
    a:Ob C
    b:Ob C
```

```
    h:Hom C a b
    {
      k:Hom C d a
      _ =  pr_assoc C r d a b h k f
    }
    Cf_IsNatural = eqExt (o SET (homr_ob C d a) (homr_ob C r a) (homr_ob C r b)
                                (homr_mor C r a b h) (Cf_ a))
                         (o SET (homr_ob C d a) (homr_ob C d b) (homr_ob C r b)
                                (Cf_ b) (homr_mor C d a b h))
                         _
  }
  Cf_Nat = (Cf_, Cf_IsNatural) ∈ NatTrans C SET (homr_ C d) (homr_ C r)
  tau:NatTrans C SET (homr_ C r) K
  ;_ o C[f,_]
  _oCf_ = comp_functs C SET (homr_ C d) (homr_ C r) K tau Cf_Nat
}
;;;Psi(d,K)• (_ o C[f,_]) = K(f)• Psi(r,K)
{
  r:Ob C
  d:Ob C
  K:Functor C SET
  f:Hom C r d
  {
    phi:NatTrans C SET (homr_ C r) K
    _1 = eqEq (eqTrans (pr_idr C d r f) (pr_idl C r d f)) (π1 phi d)
    _2 = eqSym (eqEq (π2 phi r d f)
                     (λf:(π1 (homr_ C r) r→π1 K d).f (id C r)))
    _3 = eqTrans _1 _2
  }
  PsiNatinr = eqExt (Map_comp (Psi d K) (_oCf_ r d K f))
                    (Map_comp ((π1 π2 K r d) f) (Psi r K))
                    _3
}
PsiNatinr ∈ Πr:Ob C.Πd:Ob C.ΠK:Functor C SET.Πf:Hom C r d.
                Map_comp (Psi d K) (_oCf_ r d K f)
              = Map_comp ((π1 π2 K r d) f) (Psi r K)
              ∈ Map (NatTrans C SET (Homr_ C r) K) (π1 K d)
```

```
  {
    K:Functor C SET
    H:Functor C SET
    r:Ob C
    mu:NatTrans C SET K H
    tau:NatTrans C SET (homr_ C r) K
    muo_ = comp_functs C SET (homr_ C r) K H mu tau ∈ NatTrans C SET (homr_ C r) H
  }
  ;Psi(r,H)• (_ o mu) = mu.r• Psi(r,K)
  {
    K:Functor C SET
    H:Functor C SET
    r:Ob C
    mu:NatTrans C SET K H
    {
      phi:NatTrans C SET (homr_ C r) K
      _ = eqRef
          ∈ ((π1 mu r) ((π1 phi r) (id C r)) = (π1 mu r) ((π1 phi r) (id C r))
                                                ∈ (π1 H r))
    }
    PsiNatinK = eqExt (Map_comp (Psi r H) (muo_ K H r mu))
                      (Map_comp (π1 mu r) (Psi r K))
                      _
  }
  PsiNatinK ∈ ΠK:Functor C SET.ΠH:Functor C SET.
              Πr:Ob C.Πmu:NatTrans C SET K H.
              Map_comp (Psi r H) (muo_ K H r mu) = Map_comp (π1 mu r) (Psi r K)
                 ∈ Map (NatTrans C SET (homr_ C r) K) (π1 H r)
}
```

Appendix C

Formalization T2

```
;;;preliminaries for setoid
U = Type(i)
Rel = λS:U.S → S → Prop
{
  S: U
  R: Rel S
  Refl = Πx:S.R x x
  Sym = Πx:S.Πy:S.R x y → R y x
  Trans = Πx:S.Πy:S.Πz:S.R x y → R y z → R x z
  Equiv = Refl ∧ Sym ∧ Trans
}
Setoid(i) = ΣS:U.ΣR:Rel S.Equiv S R
;;;Selectors for Setoid
{
  A:Setoid(i)
  elem = π 1 A
  equal = π1 π2 A
  pr_equiv = π2 π2 A
  pr_refl = andEL (π2 π2 A)
  pr_sym = λx|elem A.λy|elem A.λp:equal A x y.andEL (andER (π2 π2 A)) x y p
  pr_trans = λx|elem A.λy|elem A.λz|elem A.
             λp:equal A x y.λq:equal A y z.andER (andER (π2 π2 A)) x y z p q
}

;;;the type Map_setoid of setoids of Maps
{
  A:Setoid(i)
```

```
  B:Setoid(i)
  Map_law = λf:elem A → elem B.Πx:elem A.Πy:elem A.
    equal A x y → equal B (f x) (f y)
  Map = Σf:elem A → elem B.Map_law f ∈ Type0
  ap = λm:Map.π1 m
  ext = λf:Map.λg:Map.Πx:elem A.equal B (ap f x) (ap g x)
  #define ≡ ext LEFT
  {
    extIsRefl = λf:Map.λx:elem A.pr_refl B (ap f x)
    extIsSym = λf:Map.λg:Map.λe:f ≡ g.λx:elem A.pr_sym B (e x)
    extIsTrans = λf:Map.λg:Map.λh:Map.λe1:f ≡ g.λe2:g ≡ h.
                 λx:elem A.pr_trans B (e1 x) (e2 x)
    Equiv_map_eq = andI extIsRefl (andI extIsSym extIsTrans)
  }
  Map_setoid(i) = (Map, ext, Equiv_map_eq) ∈ Setoid(i)
}
ap2 = λA:Setoid(i).λB:Setoid(i).λC:Setoid(i).
      λf:elem (Map_setoid A (Map_setoid B C)).
      λa:elem A.ap B C (ap A (Map_setoid B C) f a)

;;;Definition of the type Cat
{
  Ob:Type(i)
  H:Ob → Ob → Setoid(j)
  Comp_Type = Πa:Ob.Πb:Ob.Πc:Ob.
              elem (Map_setoid (H b c) (Map_setoid (H a b) (H a c)))
  Id_Type = Πa:Ob.elem (H a a)
  o:Comp_Type
  comp = λa:Ob.λb:Ob.λc:Ob.ap2 (H b c) (H a b) (H a c) (o a b c)
  Assoc_law = Πa:Ob.Πb:Ob.Πc:Ob.Πd:Ob.
              Πf:elem(H c d).Πg:elem (H b c).Πh:elem (H a b).
              equal (H a d) (comp a c d f (comp0 a b c g h))
                            (comp a b d (comp0 b c d f g) h)
  id:Id_Type
  Idl_law = Πa:Ob.Πb:Ob.Πf:elem (H a b).
    equal (H a b) f (comp a a b f (id a))
  Idr_law = Πa:Ob.Πb:Ob.Πf:elem (H b a).
```

```
    equal (H b a) (comp b a a (id a) f) f
}
Cat(i,j) = Σ Ob:Type0.Σ Hom:Ob → Ob → Setoid0.
           Σ o:Comp_Type Ob Hom.Σ id:Id_Type Ob Hom.
           (Assoc_law Ob Hom o) × (Idl_law Ob Hom o id) × (Idr_law Ob Hom o id)
;;;Selectors for Cat
{
  C:Cat(i,j)
  Ob = π1 C
  Hom = π1 π2 C
  o = π1 π2 π2 C
  id = π1 π2 π2 π2 C
  pr_assoc = π1 π2 π2 π2 π2 C
  pr_idl = π1 π2 π2 π2 π2 π2 C
  pr_idr = π2 π2 π2 π2 π2 π2 C
}
;;;C is locally small ⇔ C:Cat(X,0)

;;;Definition of the type Functor of (covariant) functors
{
  C:Cat(i,j)
  D:Cat(k,l)
  {
    Fob:Ob C → Ob D
    Fmor:Πa:Ob0 C.Πb:Ob0 C.Map (Hom C a b) (Hom D (Fob a) (Fob b))
    compC = comp (Ob C) (Hom C) (o C)
    compD = comp (Ob D) (Hom D) (o D)
    Comp_law = Πa:Ob C.Πb:Ob C.Πc:Ob C.
               Πf:elem (Hom C b c).Πg:elem (Hom C a b).
               equal (Hom D (Fob a) (Fob c))
              (π1 (Fmor a c) (compC a b c f g))
                      (compD (Fob a) (Fob b) (Fob c)
                             (π1 (Fmor b c) f) (π1 (Fmor a b) g))
    Id_law = Πa:Ob C.equal (Hom D (Fob a) (Fob a))
                           (π1 (Fmor a a) (id C a)) (id D (Fob a))
  }
  Functor = Σ Fob:Ob C → Ob D.
```

```
            Σ Fmor:Π a:Ob C.Π b:Ob C.Map (Hom C a b) (Hom D (Fob a) (Fob b)).
            (Comp_law Fob Fmor) × (Id_law Fob Fmor)
}

;;;Definition the type ContraFunctor of contravariant functors
{
  C:Cat(i,j)
  D:Cat(k,l)
  {
    Fob:Ob C → Ob D
    Fmor:Π a:Ob C.Π b:Ob C.Map (Hom C a b) (Hom D (Fob b) (Fob a))
    compC = comp0 (Ob C) (Hom C) (o C)
    compD = comp1 (Ob D) (Hom D) (o D)
    coComp_law = Π a:Ob C.Π b:Ob C.Π c:Ob C.
                 Π g:elem (Hom C b c).Π f:elem (Hom C a b).
                 equal (Hom D (Fob c) (Fob a))
                       (π1 (Fmor a c) (compC a b c g f))
                       (compD (Fob c) (Fob b) (Fob a)
                              (π1 (Fmor a b) f) (π1 (Fmor b c) g))
    coId_law = Π a:Ob C.equal (Hom D (Fob a) (Fob a))
                              (π1 (Fmor a a) (id C a)) (id D (Fob a))
  }
  ContraFunctor = Σ Fob:Ob C → Ob D.
                  Σ Fmor:Π a:Ob C.Π b:Ob C.Map (Hom C a b) (Hom D (Fob b) (Fob a)).
                  (coComp_law Fob Fmor) × (coId_law Fob Fmor)
}

;;;Def. of Natural transformation
{
  C:Cat(i,j)
  D:Cat(k,l)
  cmp = comp (Ob D) (Hom D) (o D)
  F:Functor C D
  G:Functor C D
  NatTrans = Σ tau:Π a:Ob C.elem (Hom D (π1 F a) (π1 G a)).
             Π a:Ob C.Π b:Ob C.Π f:elem (Hom C a b).
             equal (Hom D (π1 F a) (π1 G b))
```

```
          (cmp (π1 F a) (π1 G a) (π1 G b)
                       (π1 (π1 π2 G a b) f) (tau a))
                  (cmp (π1 F a) (π1 F b) (π1 G b)
                       (tau b) (π1 (π1 π2 F a b) f))
}

;;;category SET of all sets
{
  a:Setoid(0)
  b:Setoid(0)
  c:Setoid(0)
  phi:Map b c
  psi:Map a b

  phi_psi = λx:elem a.ap b c phi (ap a b psi x)
  phi_psiIsMap = λx:elem a.λy:elem a.λe:equal a x y.
      π2 phi (ap a b psi x) (ap a b psi y) (π2 psi x y e)

  Map_comp = (phi_psi,phi_psiIsMap) ∈ Map a c
}
Map_comp  ∈ Πa:Setoid(0).Πb:Setoid(0).Πc:Setoid(0).Map b c → Map a b → Map a c
{
  a:Setoid(0)
  b:Setoid(0)
  c:Setoid(0)
  phi:Map b c
  phi1 = Map_comp a b c phi
  {
    f:Map a b
    g:Map a b
    e:ext a b f g
    z:elem a
    p = π2 phi (ap a b f z) (ap a b g z) (e z)
  }
  Map_comp1 = (phi1, p) ∈ (Map (Map_setoid a b) (Map_setoid a c))
}
Map_comp1 ∈ Πa:Setoid(0).Πb:Setoid(0).Πc:Setoid(0).
```

```
            Map b c → Map (Map_setoid a b) (Map_setoid a c)
{
  a:Setoid(0)
  b:Setoid(0)
  c:Setoid(0)
  phi =  Map_comp1 a b c
  {
    f:Map b c
    g:Map b c
    e:ext b c f g
    F:Map a b
    z:elem a
    p = e (π1 F z)
  }
  comp_set = (phi,p) ∈ Map (Map_setoid b c)
                           (Map_setoid (Map_setoid a b) (Map_setoid a c))
}
comp_set ∈ Comp_Type Setoid(0) Map_setoid(0)
{
  a:Setoid(0)
  b:Setoid(0)
  c:Setoid(0)
  d:Setoid(0)
  f:Map c d
  g:Map b c
  h:Map a b
  x:elem a
  assoc_set = π2 f (π1 g (π1 h x)) (π1 g (π1 h x))
                   (π2 g (π1 h x) (π1 h x) (π2 h x x (pr_refl a x)))
}
assoc_set ∈ Assoc_law Setoid(0) Map_setoid(0) comp_set
{
  A:Setoid(0)
  id_set = ((λx:elem A.x), (λx:elem A.λy:elem A.λe:equal A x y.e)) ∈ Map A A
}
id_set ∈ ΠA:Setoid(0).Map A A
{
```

```
  a:Setoid(0)
  b:Setoid(0)
  f:elem (Map_setoid a b)
  x:elem a
  idl_set = pr_refl b (π1 f x)
}
idl_set  ∈ Idl_law Setoid(0) Map_setoid(0) comp_set id_set
{
  a:Setoid(0)
  b:Setoid(0)
  f:Map b a
  x:elem b
  idr_set = pr_refl a (π1 f x)
}
idr_set  ∈ Idr_law Setoid(0) Map_setoid(0) comp_set id_set
SET = (Setoid(0), Map_setoid(0), comp_set, id_set, assoc_set, idl_set, idr_set) ∈ Cat(1,0)

;;;functor category
{
  A:Cat(i,j)
  B:Cat(k,l)
  NatTr = NatTrans A B
  compB = comp (Ob B) (Hom B) (o B)
  Functs = Functor A B
  {
    F:Functs
    G:Functs
    NatTrFG = NatTr F G
    natext = λt1:NatTrFG.λt2:NatTrFG.
             Πa:Ob A.equal (Hom B (π1 F a) (π1 G a)) (π1 t1 a) (π1 t2 a)
    #define ≡ natext LEFT
    natextIsRefl = λt:NatTrFG.
                   λa:Ob A.pr_refl (Hom B (π1 F a) (π1 G a)) (π1 t a)
    natextIsSym = λt1:NatTrFG.λt2:NatTrFG.λe:t1 ≡ t2.
                  λa:Ob A.pr_sym (Hom B (π1 F a) (π1 G a)) (e a)
    natextIsTrans = λt1:NatTrFG.λt2:NatTrFG.λt3:NatTrFG.λe1:t1 ≡ t2.λe2:t2 ≡ t3.
                    λa:Ob A.pr_trans (Hom B (π1 F a) (π1 G a)) (e1 a) (e2 a)
```

```
  Equiv_nat_eq = andI natextIsRefl (andI natextIsSym natextIsTrans)
  Nat_setoid = (NatTrFG, natext, Equiv_nat_eq) ∈ Setoid(0)
}
{
  F1:Functs
  F2:Functs
  F3:Functs
  t1:NatTr F2 F3
  t2:NatTr F1 F2
  t1t2 = λa:Ob A.compB (π1 F1 a) (π1 F2 a) (π1 F3 a) (π1 t1 a) (π1 t2 a)
  {
    a:Ob A
    b:Ob A
    f:elem (Hom A a b)
    hom_setoid = (Hom B (π1 F1 a) (π1 F3 b))
    ;F3(f) • (t1(a)) • t2(a))=(t1(b) • t2(b)) • F1(f)
    {
      ;F3(f) • t1(a)=t1(b) • F2(f)
      t1Isnat=π2 t1 a b f
      ;F2(f) • t2(a)=t2(b) • F1(f)
      t2Isnat=π2 t2 a b f
      ;F3(f) • (t1(a)) • t2(a))=(F3(f) • t1(a)) • t2(a)
      _0 = pr_assoc D (π1 F1 a) (π1 F2 a) (π1 F3 a) (π1 F3 b)
                      (π1 (π1 π2 F3 a b) f) (π1 t1 a) (π1 t2 a)
      ;t1(b) • (F2(f) • t2(a))=(t1(b) • F2(f)) • t2(a)
      _1 = pr_assoc D (π1 F1 a) (π1 F2 a) (π1 F2 b) (π1 F3 b)
                      (π1 t1 b) (π1 (π1 π2 F2 a b) f) (π1 t2 a)
      ;t1(b) • (t2(b) • F1(F))=(t1(b) • t2(b)) • F1(F)
      _2 = pr_assoc D (π1 F1 a) (π1 F1 b) (π1 F2 b) (π1 F3 b)
                      (π1 t1 b) (π1 t2 b) (π1 (π1 π2 F1 a b) f)
      t1xt2Isnat = π2 (π1 (o B (π1 F1 a) (π1 F2 b) (π1 F3 b)) (π1 t1 b))
                       (compB (π1 F1 a) (π1 F2 a) (π1 F2 b)
                              (π1 (π1 π2 F2 a b) f) (π1 t2 a))
                       (compB (π1 F1 a) (π1 F1 b) (π1 F2 b) (π1 t2 b)
                              (π1 (π1 π2 F1 a b) f))
                       t2Isnat
      t1Isnatxt2 = π2 (o B (π1 F1 a) (π1 F2 a) (π1 F3 b))
```

```
                    (compB (π1 F2 a) (π1 F3 a) (π1 F3 b)
                           (π1 (π1 π2 F3 a b) f) (π1 t1 a))
                    (compB (π1 F2 a) (π1 F2 b) (π1 F3 b) (π1 t1 b)
                           (π1 (π1 π2 F2 a b) f))
                    t1Isnat
                    (π1 t2 a)
      t1t2IsNatural = pr_trans hom_setoid
                               _0
                               (pr_trans hom_setoid
                                         t1Isnatxt2
                                         (pr_trans hom_setoid
                                                   (pr_sym hom_setoid_1)
                                                   pr_trans hom_setoid
                                                            t1xt2Isnat
                                                            _2)
    }
  }
  nat_comp = (t1t2, t1t2IsNatural) ∈ NatTr F1 F3
}
nat_comp ∈ ΠF1:Functs.ΠF2:Functs.ΠF3:Functs.
            NatTr F2 F3 → NatTr F1 F2 → NatTr F1 F3
{
  F1:Functs
  F2:Functs
  F3:Functs
  t:NatTr F2 F3
  tmap = nat_comp F1 F2 F3 t ∈ NatTr F1 F2 → NatTr F1 F3
  {
    t1:elem (Nat_setoid F1 F2)
    t2:elem (Nat_setoid F1 F2)
    eq0:equal (Nat_setoid F1 F2) t1 t2
    a:Ob A
    tmapIsmap = π2 (π1 (o B (π1 F1 a) (π1 F2 a) (π1 F3 a)) (π1 t a))
                (π1 t1 a) (π1 t2 a) (eq0 a)
  }
  nat_comp1 = (tmap, tmapIsmap) ∈ Map (Nat_setoid F1 F2) (Nat_setoid F1 F3)
}
```

```
nat_comp1 ∈ ΠF1:Functs.ΠF2:Functs.ΠF3:Functs.
            NatTr F2 F3 → Map (Nat_setoid F1 F2) (Nat_setoid F1 F3)
{
  F1:Functs
  F2:Functs
  F3:Functs
  o123 = nat_comp1 F1 F2 F3 ∈ NatTr F2 F3 →
                              Map (Nat_setoid F1 F2) (Nat_setoid F1 F3)
  {
    t1:NatTr F2 F3
    t2:NatTr F2 F3
    e:equal (Nat_setoid F2 F3) t1 t2
    t:NatTr F1 F2
    a:Ob A
    o123Map = π2 (o B  (π1 F1 a) (π1 F2 a) (π1 F3 a))
              (π1 t1 a) (π1 t2 a) (e a) (π1 t a)
  }
  comp_functs = (o123, o123Map) ∈ Map (Nat_setoid F2 F3)
                                      (Map_setoid (Nat_setoid F1 F2)
                                                  (Nat_setoid F1 F3))
}
comp_functs ∈ Comp_Type Functs Nat_setoid
{
  F1:Functs
  F2:Functs
  F3:Functs
  F4:Functs
  t1:NatTr F3 F4
  t2:NatTr F2 F3
  t3:NatTr F1 F2
  a:Ob A
  assoc_functs = pr_assoc (π1 F1 a) (π1 F2 a) (π1 F3 a) (π1 F4 a)
                          (π1 t1 a) (π1 t2 a) (π1 t3 a)
}
assoc_functs ∈ Assoc_law Functs Nat_setoid comp_functs
{
  F:Functs
```

```
    iota = λa:Ob A.(id B (π1 F a)) ∈ Πa:π1 A.π1 (π1 π2 B (π1 F a) (π1 F a))
    {
      a:Ob A
      b:Ob A
      f:elem (Hom A a b)
      iotaNat = pr_sym (Hom B (π1 F a) (π1 F b))
                       (pr_trans (Hom B (π1 F a) (π1 F b))
                           (pr_idr D (π1 F b) (π1 F a) (π1 (π1 π2 F a b) f))
                           (pr_idl D (π1 F a) (π1 F b) (π1 (π1 π2 F a b) f)))
    }
    id_functs = (iota, iotaNat) ∈ NatTr F F
  }
  id_functs ∈ Id_Type Functs Nat_setoid
  {
    F1:Functs
    F2:Functs
    t:NatTr F1 F2
    a:Ob A
    idl_functs = pr_idl D (π1 F1 a) (π1 F2 a) (π1 t a)
  }
  idl_functs ∈ Idl_law Functs Nat_setoid comp_functs id_functs
  {
    F1:Functs
    F2:Functs
    t:NatTr F2 F1
    a:Ob A
    idr_functs = pr_idr D (π1 F1 a) (π1 F2 a) (π1 t a)
  }
  idr_functs ∈ Idr_law Functs Nat_setoid comp_functs id_functs
  Funct = (Functs, Nat_setoid, comp_functs, id_functs, assoc_functs, idl_functs, idr_functs)
}
;;;hom-functors
{
  C:Cat(i,0)
  r:Ob C
  oC = comp (Ob C) (Hom C) (o C)
  homr_ob = λa:Ob C.Hom C r a
```

```
{
  a:Ob C
  b:Ob C
  {
    f:elem (Hom C a b)
    phi = λg:elem (Hom C r a).oC r a b f g
    {
      g1:elem (Hom C r a)
      g2:elem (Hom C r a)
      e:equal (Hom C r a) g1 g2
      phiIsMap = π2 (π1 (o C r a b) f) g1 g2 e
    }
    F2map = (phi, phiIsMap) ∈ Map (homr_ob a) (homr_ob b)
  }
  {
    f1:elem (Hom C a b)
    f2:elem (Hom C a b)
    e0:equal (Hom C a b) f1 f2
    g:elem (Hom C r a)
    F2mapIsMap = π2 (o C r a b) f1 f2 e0 g
  }
  homr_mor = (F2map, F2mapIsMap) ∈ Map (Hom C a b)
                                       (Hom SET (homr_ob a) (homr_ob b))
}
pr_assoc = π1 π2 π2 π2 π2 C
{
  a:Ob C
  b:Ob C
  c:Ob C
  f:elem (Hom C b c)
  g:elem (Hom C a b)
  h:elem (Hom C r a)
  pr_comp = pr_sym (Hom C r c) (pr_assoc r a b c f g h)
}
pr_id = λa:Ob C.λf:elem(Hom C r a).pr_idr C a r f
;covariant hom-functor
homr_ = (homr_ob, homr_mor, pr_comp, pr_id) ∈ Functor C SET
```

```
hom_rob = λa:Ob C.Hom C a r
{
  a:Ob C
  b:Ob C
  {
    f:elem (Hom C a b)
    psi = λh:elem (Hom C b r).oC a b r h f
    {
      h1:elem (Hom C b r)
      h2:elem (Hom C b r)
      e:equal (Hom C b r) h1 h2
      psiIsMap = π2 (o C a b r) h1 h2 e f
    }
    F2map = (psi, psiIsMap) ∈ Map (hom_rob b) (hom_rob a)
  }
  {
    f1:elem(Hom C a b)
    f2:elem (Hom C a b)
    e0:equal (Hom C a b) f1 f2
    h:elem (Hom C b r)
    F2mapIsMap = π2 (π1 (o C a b r) h) f1 f2 e0
  }
  hom_rmor = (F2map, F2mapIsMap) ∈ Map (Hom C a b)
                                       (Hom SET (Hom C b r) (Hom C a r))
}
{
  a:Ob C
  b:Ob C
  c:Ob C
  g:elem (Hom C b c)
  f:elem (Hom C a b)
  h:elem (Hom C c r)
  pr_cocomp = pr_assoc a b c r h g f
}
pr_coid = λa:Ob C.λf:elem (Hom C a r).pr_sym (Hom C a r) (pr_idl C a r f)
;contravariant hom-functor
```

```
  hom_r = (hom_rob, hom_rmor, pr_cocomp, pr_coid) ∈ ContraFunctor C SET
}

;;;Yoneda
{
  C:Cat(i,0)
  comp=λa:Ob C.λb:Ob C.λc:Ob C.
       ap2 (Hom C b c) (Hom C a b) (Hom C a c) (o C a b c)
  {
    r:Ob C
    K:Functor C SET
    {
      phi:NatTrans C SET (homr_ C r) K
      d:Ob C
      f:elem (Hom C r d)
      {
        e1 = π2 (π1 phi d) f (comp r r d f (id C r)) (pr_idl C r d f)
        e2 = pr_sym (π1 K d) ((π2 phi r d f) (id C r))
        YonedaDiagram = pr_trans (π1 K d) e1 e2
      }
    }
    YonedaDiagram ∈ Πphi:NatTrans C SET (homr_ C r) K.
                    Πd:Ob C.Πf:elem(Hom C r d).
                    equal (π1 K d)
                          (π1 (π1 phi d) f)
                          (π1 (π1 (π1 π2 K r d) f) (π1 (π1 phi r) (id C r)))
    {
      phi:NatTrans C SET (homr_ C r) K
      Psi = π1 (π1 phi r) (id C r)
    }
    {
      phi1:NatTrans C SET (homr_ C r) K
      phi2:NatTrans C SET (homr_ C r) K
      e0:equal (Nat_setoid C SET (homr_ C r) K) phi1 phi2
      PsiIsMap = e0 r (id C r)
    }
    PsiMap = (Psi, PsiIsMap) ∈ Map (Nat_setoid C SET (homr_ C r) K) (π1 K r)
```

```
{
  phi1:NatTrans C SET (homr_ C r) K
  phi2:NatTrans C SET (homr_ C r) K
  e0:equal (π1 K r) (Psi phi1) (Psi phi2)
  d:Ob C
  f:elem(Hom C r d)
  {
    e1 = YonedaDiagram phi1 d f
    e2 = pr_sym (π1 K d) (YonedaDiagram phi2 d f)
    e3 = π2 (π1 (π1 π2 K r d) f) (Psi phi1) (Psi phi2) e0
    PsiInjective = pr_trans (π1 K d) e1 (pr_trans (π1 K d) e3 e2)
  }
}
PsiInjective ∈ Πphi1:NatTrans C SET (homr_ C r) K.
               Πphi2:NatTrans C SET (homr_ C r) K.
               equal (π1 K r) (Psi phi1) (Psi phi2) →
               Πd:Ob C.Πf:elem(Hom C r d).
               equal (π1 K d) (π1 (π1 phi1 d) f) (π1 (π1 phi2 d) f)
{
  x:elem (π1 K r)
  phi0 = λd:Ob C.λf:elem(Hom C r d).π1 (π1 (π1 π2 K r d) f) x
  {
    a:Ob C
    m = phi0 a  ∈ elem (Hom C r a) → elem (π1 K a)
    {
      f:elem (Hom C r a)
      g:elem (Hom C r a)
      e:equal (Hom C r a) f g
      mIsMap = π2 (π1 π2 K r a) f g e x
    }
    phi = (m, mIsMap) ∈ Map (Hom C r a) (π1 K a)
  }

  {
    a:Ob C
    b:Ob C
    f:elem (Hom C a b)
```

```
      g:elem (Hom C r a)
      phiIsNatural = pr_sym (π1 K b) (π1 π2 π2 K r a b f g x)
    }
    phiNat = (phi, phiIsNatural) ∈ NatTrans C SET (homr_ C r) K

    PsiSurjective = exI phiNat (π2 π2 π2 K r x)
       ∈ ∃ phiNat:NatTrans C SET (homr_ C r) K.equal (π1 K r) (Psi phiNat) x
  }
  PsiSurjective ∈ Πx:elem(π1 K r). ∃ phiNat:NatTrans C SET (homr_ C r) K.
                  equal (π1 K r) (Psi phiNat) x
  IsoType = (Πphi1:NatTrans C SET (homr_ C r) K.
             Πphi2:NatTrans C SET (homr_ C r) K.
             equal (π1 K r) (Psi phi1) (Psi phi2) →
             Πd:Ob C.Πf:elem(Hom C r d).
             equal (π1 K d) (π1 (π1 phi1 d) f) (π1 (π1 phi2 d) f))
          ∧ (Πx:elem(π1 K r). ∃ phiNat:NatTrans C SET (homr_ C r) K.
             equal (π1 K r) (Psi phiNat) x)
  YonedaLemma = andI PsiInjective PsiSurjective ∈ IsoType
}
YonedaLemma ∈ Πr:Ob C.ΠK:Functor C SET.IsoType r K
;;;Naturality of Psi in r
{
  r:Ob C
  d:Ob C
  K:Functor C SET
  f:elem(Hom C r d)
  Cf_ = λc:Ob C.π1 (hom_rmor C c r d) f  ∈ Πc:Ob C.Map (Hom C d c) (Hom C r c)
  {
    a:Ob C
    b:Ob C
    h:elem (Hom C a b)
    k:elem (Hom C d a)
    Cf_IsNatural = pr_assoc C r d a b h k f
  }
  ;natural transformantion C[f,_]:C[d,_] → C[r,_]
  Cf_Nat = (Cf_, Cf_IsNatural) ∈ NatTrans C SET (homr_ C d) (homr_ C r)
  {
```

```
    tau:NatTrans C SET (homr_ C r) K
    ;_ o C[f,_]
    _oCf_ = nat_comp C SET (homr_ C d) (homr_ C r) K tau Cf_Nat
  }
  {
    phi1:NatTrans C SET (homr_ C r) K
    phi2:NatTrans C SET (homr_ C r) K
    e0:equal (Nat_setoid C SET (homr_ C r) K) phi1 phi2
    a:Ob C
    g:elem (Hom C d a)
    _oCf_IsMap = e0 a (π1 (π1 (o C r d a) g) f)
  }
  _oCf_Map = (_oCf_, _oCf_IsMap) ∈ Map (Nat_setoid C SET (homr_ C r) K)
                                       (Nat_setoid C SET (homr_ C d) K)
}
;;;Psi(d,K)• (_ o C[f,_]) = K(f)• Psi(r,K)
{
  r:Ob C
  d:Ob C
  K:Functor C SET
  f:elem (Hom C r d)
  phi:elem (Nat_setoid C SET (homr_ C r) K)
  {
    e1 = pr_trans (Hom C r d) (pr_idr C d r f) (pr_idl C r d f)
    e2 = π2 (π1 phi d) (comp r d d (id C d) f) (comp r r d f (id C r)) e1
    e3 = pr_sym (π1 K d) (π2 phi r d f (id C r))
    PsiNatinr = pr_trans (π1 K d) e2 e3
  }
}
PsiNatinr ∈ Πr:Ob C.Πd:Ob C.ΠK:Functor C SET.Πf:elem (Hom C r d).
            equal (Map_setoid (Nat_setoid C SET (homr_ C r) K) (π1 K d))
                  (Map_comp (Nat_setoid C SET (homr_ C r) K)
                            (Nat_setoid C SET (homr_ C d) K)
                            (π1 K d)
                            (PsiMap d K) (_oCf_Map r d K f))
                  (Map_comp (Nat_setoid C SET (homr_ C r) K)
                            (π1 K r)
```

```
                              (π1 K d)
                              (π1 (π1 π2 K r d) f) (PsiMap r K))
;;;Naturality of Psi in K
{
  K:Functor C SET
  H:Functor C SET
  r:Ob C
  mu:NatTrans C SET K H
  {
    tau:NatTrans C SET (homr_ C r) K
    muo_ = nat_comp C SET (homr_ C r) K H mu tau ∈ NatTrans C SET (homr_ C r) H
  }
  {
    tau1:NatTrans C SET (homr_ C r) K
    tau2:NatTrans C SET (homr_ C r) K
    e0:equal (Nat_setoid C SET (homr_ C r) K) tau1 tau2
    a:Ob C
    g:elem (Hom C r a)
    muo_IsMap = π2 (π1 mu a) (π1 (π1 tau1 a) g) (π1 (π1 tau2 a) g) (e0 a g)
  }
  muo_Map = (muo_, muo_IsMap) ∈ Map (Nat_setoid C SET (homr_ C r) K)
                                    (Nat_setoid C SET (homr_ C r) H)
}
;Psi(r,H)• (_ o mu) = mu.r• Psi(r,K)
{
  K:Functor C SET
  H:Functor C SET
  r:Ob C
  mu:NatTrans C SET K H
  phi:NatTrans C SET (homr_ C r) K
  PsiNatinK = pr_refl (π1 H r) (π1 (π1 mu r) (π1 (π1 phi r) (id C r)))
}
PsiNatinK ∈ ΠK:Functor C SET.ΠH:Functor C SET.Πr:Ob C.Πmu:NatTrans C SET K H.
             equal (Map_setoid (Nat_setoid C SET (homr_ C r) K) (π1 H r))
                   (Map_comp (Nat_setoid C SET (homr_ C r) K)
                             (Nat_setoid C SET (homr_ C r) H)
                             (π1 H r)
```

```
                    (PsiMap r H) (muo_Map K H r mu))
                (Map_comp (Nat_setoid C SET (homr_ C r) K)
                    (π1 K r)
                    (π1 H r)
                    (π1 mu r) (PsiMap r K))
}
```