The IFF Foundation Ontology

"Philosophy cannot become scientifically healthy without an immense technical vocabulary. We can hardly imagine our great-grandsons turning over the leaves of this dictionary without amusement over the paucity of words with which their grandsires attempted to handle metaphysics and logic. Long before that day, it will have become indispensably requisite, too, that each of these terms should be confined to a single meaning which, however broad, must be free from all vagueness. This will involve a revolution in terminology; for in its present condition a philosophical thought of any precision can seldom be expressed without lengthy explanations." – Charles Sanders Peirce, Collected Papers 8:169

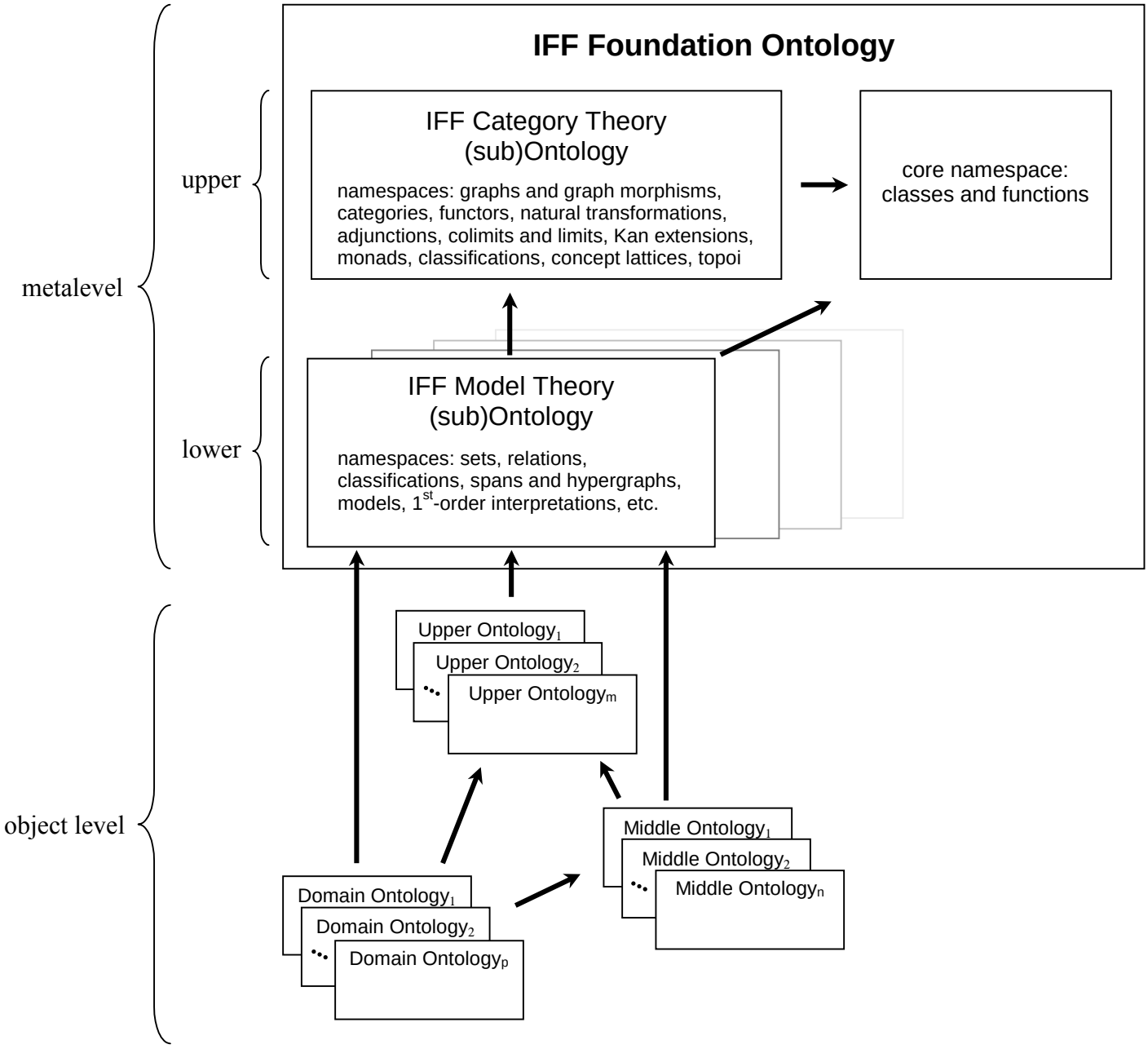


Figure 1: IFF Foundation Ontology Architecture (with dependencies)

[The numbering of figures, diagrams and tables restarts in each top-level namespace.]

Introduction

The Information Flow Framework (IFF)

The mission of the Information Flow Framework (IFF) is to further the development of the theory of Information Flow, and to apply Information Flow to distributed logic, ontologies, and knowledge representation. IFF provides mechanisms for a principled foundation for an ontological framework – a framework for sharing ontologies, manipulating ontologies as objects, partitioning ontologies, composing ontologies, discussing ontological structure, noting dependencies between ontologies, declaring the use of other ontologies, etc. IFF is *primarily* based upon the theory of Information Flow initiated by Barwise (Barwise and Seligman 1997), which is centered on the notion of a *classification*. Information Flow itself based upon the theory of the Chu construction of $*$-autonomous categories (Barr 1996), thus giving it a connection to concurrency and Linear Logic. IFF is *secondarily* based upon the theory of Formal Concept Analysis initiated by Wille (Ganter & Wille 1999), which is centered on the notion of a *concept lattice*. IFF represents metalogic, and as such operates at the structural level of ontologies. In IFF there is a precise boundary between the metalevel and the object level. The structure of IFF is illustrated in Figure 1.

Architecture

This section describes an overall scheme for an "industrial strength" ontological framework for the SUO, which represents the SUO structural level in terms of IFF meta-ontologies.

At the highest level of the SUO is the *Basic KIF Ontology*, whose purpose is to provide an interface between the new KIF and the SUO ontological structure. The Basic KIF Ontology provides an adequate foundation for representing ontologies in general and for defining the other metalevel ontologies (Figure 1) in particular. All upper metalevel ontologies import and use, either directly or indirectly, the Basic KIF Ontology.

The IFF Foundation Ontology represents the *structural aspect* of the SUO called the metalevel. The IFF Foundation Ontology is rather large, but is highly structured. It is partitioned into two levels (upper/lower) corresponding to the large/small distinction in foundations – the upper level is for set-theoretically large notions and the lower level is for set-theoretically small notions. Each level is divided into about ten namespaces. The structure of the IFF Foundation Ontology (Figure 1) is as follows.

1. Upper metalevel
   a. core namespace of set-theoretic classes and their functions
   b. Category Theory Ontology (~10 namespaces)
2. Lower metalevel
   a. Model Theory Ontology (~10 namespace)
   b. …

The upper metalevel has a distinguished namespace (SET) for set-theoretic classes and their functions called the *core namespace*. The remainder of the upper metalevel, called the *IFF Category Theory Ontology*, represents in various namespaces standard ideas of category theory. Much of the content comes from well-known category-theoretic intuitions and some of the semi-standard notation used in papers and textbooks. The IFF Category Theory Ontology provides a framework for reasoning about metalogic, as represented in the lower metalevel ontologies. Examples of provable statements are: "the Classification namespace represents a category," or "the IF classification functor is left adjoint to the IF theory functor" (Kent 2000). The IFF Category Theory Ontology is a KIF formalism for *category theory* in one of its normal presentations. Other presentations, such as home-set or arrows-only, may also have merit. The IFF Category Theory Ontology is based upon the namespace for large graphs and graph morphisms that includes horizontal multiplication of graphs and a theory of coherence – a category is a monoid in the monoidal category of large graphs.

The upper metalevel is used to represent and axiomatize the lower metalevel. The notion of a *topos*, which is central to categorical foundations, is important in the IFF Foundation Ontology, where it is encountered in at least three ways: there will be a namespace (TOP) in the upper metalevel that axiomatizes a topos; the namespace (set) for small sets and their functions in the lower metalevel can be proven to be a

topos with the TOP axioms; and (see the section on Previous Foundations) the core namespace (SET) in the IFF Foundation Ontology satisfies Colin McLarty's topos axioms. The TOP namespace logically depends (see the arrows in Figure 1) upon the namespace for categories (CAT), which in turn logically depends upon the core namespace (SET). This foundational approach should answer Solomon Feferman's qualms about logical and psychological priority.

The lower metalevel contains at present one module called the *IFF Model Theory Ontology* consisting of about ten namespaces. This expression of model theory, somewhat novel since it is based upon the theory of Information Flow, is used to represent and axiomatize the object level. One main goal of the IFF Model Theory Ontology is representation of a principled approach to ontology composition using colimits. Not unconnected to this is the principled support that the IFF Model Theory Ontology gives to John Sowa's notion of "an infinite lattice of all possible theories as the theoretical foundation of the SUO." It does this by realizing another main goal – the representation of the *truth classification* and *truth concept lattice*[1]. The truth concept lattice provides "a framework that can support an open-ended collection of ontological theories (potentially infinite) organized in a lattice together with systematic metalevel techniques for moving from one to another, for testing their adequacy for any given problem, and for mixing, matching, combining, and transforming them to whatever form is appropriate for whatever problem anyone is trying to solve" (Sowa).

A specific goal of the lower metalevel is to represent some central notions of model theory – models (first-order structures), expressions and satisfaction. In the IFF approach for structuring the SUO, models are 2-dimensional structures composed of small classifications along one dimension and small hypergraphs (or spans) along the other; here classifications represent the instance-type distinction, whereas hypergraphs represent the entity-relation distinction. The lower metalevel makes heavy use of the upper metalevel – indeed, a goal in modeling the lower metalevel is to abide by the following categorical property.

> **CATEGORICAL PROPERTY:** The (new-KIF) axiomatization for the ontologies in the lower metalevel is strictly category-theoretic – all axioms are expressed in terms of category-theoretic notions, such as the composition and identity of large/small functions or the pullback of diagrams of large/small functions; [no KIF] no axioms use explicit KIF connectives or quantification; and [no basic KIF ontology] no axioms use terms from the basic KIF ontology, other than pair bracketing '`[-]`' or pair projection '`(- 1)`', '`(- 2)`'. This would seem to extend to all ontologies for true categories (not quasi-categories) – those categories whose object and morphism collections are classes (not conglomerates).

In the lower metalevel, ontologies are organized in two dimensions corresponding to this notion of IFF model. These two precise mathematical dimensions correspond to the intuitive distinctions of Heraclitus (the *physical* versus the *abstract*) and Peirce (*1st-ness*, *2nd-ness* and *3rd-ness*).

In IFF the type-token distinction looms large. Along the instance-type dimension are the *IF classification namespace* that directly represents the instance-type distinction, the *IF theory namespace* that is connected by an adjunction to the IF classification namespace (Kent 2000), and the combining *IF logic namespace*. Terminology for the classification namespace is included in the SUO Coda and SUO Modules databases. The *concept lattice namespace* represents Formal Concept Analysis. In addition to formal concepts and their lattices, this also includes the idea of a collective concept. In the entity-relation dimension are the *hypergraph namespace* that represents multivalent relations and the *language namespace* (whose presentation is a little delicate) that represents logical expressions. Also at the lower level are the *set namespace* that models the topos of small sets, the combining *model namespace* and a derivative *ontology namespace*. The latter two namespaces are related to the fundamental truth between models and expressions. In addition, the lower metalevel namespaces have sufficient morphism and colimit structure to provide a principled approach for combining of ontologies at the object level.

Other modules (namespaces or subontologies), in addition to the IFF Model Theory Ontology, are perhaps possible at the lower metalevel. These might include modules for categorical model theory, modules for modal, tense and linear logic, modules for rough and fuzzy sets, modules for semiotics, etc.

---

[1] The truth classification of a first-order language ***L*** is the meta-classification, whose instances are ***L***-structures (models), whose types are ***L***-sentences, and whose classification relation is satisfaction. In IFF the concept lattice of the truth meta-classification functions is the appropriate "lattice of ontological theories." A formal concept in this lattice has an intent that is a closed theory (set of sentences) and an extent that is the collection of all models for that theory. The theory (intent) of the join or supremum of two concepts is the closure of the intersection of the theories (conceptual intents), and the theory (intent) of the meet or infimum of two concepts is the theory of the common models.

The object level is where the content ontologies reside. These could be very generic, such as a 4D ontology, or specific, such as ontologies for government or higher education. It should be noted that the object level satisfies a representation property similar to the categorical property satisfied by the lower metalevel – the ontological language used is not KIF; instead it is the terminology defined and axiomatized in the lower structural level (terms such as 'subtype', 'instance', 'expression', 'model', 'ontology', 'relation', 'entity', 'role', etc.).

Since the IFF Foundation Ontology represents abstract semantics, many applications are possible. Two, in particular, are being actively considered:

- a representation for the categorical framework for the Specware system of the Kestrel Institute, which is based on category theory and supports creation and combination of specifications (ontology analogs) using colimits.
- a semantics for DAML+OIL, the DARPA Agent Markup Language, whose goal is to develop language and tools to facilitate the concept of the *semantic web*.

Programming Language Analogy

At the core the SUO is coded in the new KIF. As an ontological machine language, the Lisp-y style of logical expression of the new KIF is very useful, adequately expressive and conveniently terse. However, in the proposed IFF structure for the SUO the new KIF will not be used as an "industrial strength" ontological language. Instead, the SUO terminology will follow the following programming language analogy.

- The new KIF is like an *ontological machine language*.
- The Basic KIF Ontology is like an *ontological assembly language*. It provides a basic ontological apparatus for the SUO. The purpose of this kind of terminology is to be an interface between the KIF language and other ontological terminology, in general. Because of its function, the Basic KIF Ontology might also be called the bootstrap ontology. Hopefully, this will have only one namespace, perhaps with the namespace prefix '`KIF`'.
- The metalevel or structural level of the SUO, as encoded in the IFF Foundation Ontology, is like a *high level programming language* such as Lisp, Java, ML, etc. IFF is a building blocks approach to ontological structure – a rather elaborate categorical approach that uses some insight from the theory of distributed logic called Information Flow (Barwise and Seligman, 1997), and also uses ideas from the theory of Formal Concept Analysis (Ganter and Wille, 1999). The structural level of the SUO has many namespaces, such as a core namespace '`SET`', a set namespace '`set`', a category namespace '`CAT`', a categorical colimit namespace '`COL`', a classification '`cls`' namespace, an IF theory namespace '`th`', a concept lattice namespace '`cl`', a language namespace '`lang`', a model namespace '`mod`', a truth namespace for the fundamental truth classification and concept lattice, and other supporting namespaces.
- The object level of the SUO is like the various *software applications*, such as word processors, browsers, spreadsheet software, databases, etc. The ontologies in this component are specified using the terminology axiomatized in the lower structural level. The object level provides the content of the SUO coming from the various domain ontologies and philosophies with their own namespaces. The object level would include ontologies such as Casati and Varzi's ontology of holes, John Sowa's upper ontology, Barry Smith's Formal Theory of Boundaries/Objects, Borgo, Guarino, and Masolo's Formal Theory of Physical Objects, the public Cyc ontology, ontologies for organizations, ontologies for repositories, etc.

Part of the purpose of the structural level of the SUO is to interrelate the various modules at the object level. It is important that a complete distinction and an explicit boundary is kept between the object level and the metalevel. This fundamental partition must be obviously manifest in the SUO. Some ontologies at the object level may choose to represent and discuss metalevel concepts, but they would still be doing so at the object level.

Overview

The IFF Foundation Ontology consists of an adequate amount of set theory which, on the one hand, is sufficiently flexible for the categorical inquiry involved in the Information Flow Framework (IFF) but, on the other hand, is sufficiently restrictive that IFF be consistent (does not produce contradictions). The approach to foundations used here is an adaptation of that outlined in chapter 2 of the book *Abstract and Concrete Categories* (Adámek, Herrlich & Strecker 1990). The basic concepts needed are those of *sets* and *classes*. To this we add the notion of Cartesian closure and topos structure at the level of classes.

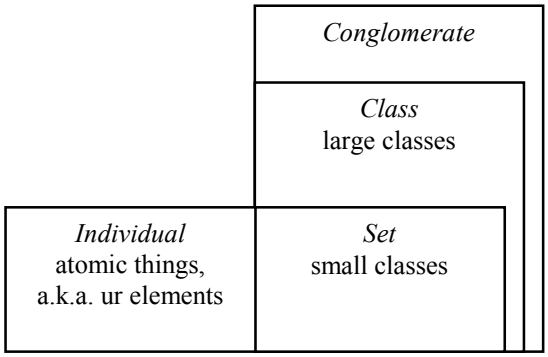


Figure 2: Collection Hierarchy

The IFF Foundation Ontology uses and imports the following terms from the Basic KIF Ontology: the KIF relational terms '`KIF$class`' (otherwise known as unary relations or predicates) and '`KIF$relation`' and '`KIF$subclass`', the KIF functional term '`KIF$function`', the generic type declaration term '`KIF$signature`', and the sequence terminology '`[-]`', '`1`', '`2`'. The term '`KIF$class`' is used in both a syntactic and a semantic sense – syntactically things that are of this type should function as KIF predicates, whereas semantically this denotes the largest kind of collection. Hence, semantically '`KIF$class`' corresponds in general to *collection* and in particular to *conglomerate*.

The IFF Foundation Ontology uses the three-level set-theoretic hierarchy of sets – classes – conglomerates. Table 1 locates various collections and functions in this three-tiered framework. Functions between conglomerates are unary or binary KIF functions. To make this the root ontology vis-à-vis importation, we have renamed these as conglomerate (CNG) functions. As CNG functions, source and target type constraints are specified with the '`CNG$signature`' relation. In contrast, functions between classes, otherwise called "SET functions," have a more abstract, semantic representation, since they have explicitly specified source and target classes and an abstract composition operation with identities. Every SET function is represented as a unary CNG function. The signature of a SET function is given by its source and target.

Table 1: Kinds of Specific Collections

Conglomerates	○ the collections of classes, functions, opspans and binary cones ○ the collections of large graphs and large graph morphisms ○ the collections of large categories and functors between large categories ○ the collection of natural transformations, and the collection of adjunctions
Classes	○ the object and morphism collections in any large category ○ the source and target of the component map of a natural transformation ○ the collections of algebras and homomorphisms of a monad ○ the collection of diagrams and cocones for each type of finite colimit in a category
Sets	○ sets as collections and the extent of their functions ○ the instance and type sets of classifications and the instance and type functions of infomorphisms ○ any small relation regarded as a collection

Previous Foundations

Topos Axioms

The core namespace in the IFF Foundation Ontology axiomatizes the quasi-category of classes and their functions. When the axioms for a topos are in place in the Category Theory Ontology, then the category of sets and their functions, which is encoded in the set namespace in the Foundation Ontology as a restriction of the core namespace, can be proven to be a well-pointed topos with natural numbers and choice. As mentioned above, the Foundation Ontology should partially satisfy Solomon Feferman's representational needs, as expressed in the FOM list thread Toposy-turvey. For purposes of comparison, and to show completeness, here is a presentation of Colin McLarty's topos axioms that give a first order expression for the theory of a *well-pointed topos with natural numbers and choice*. McLarty makes the following claims.

- The theory was given by Lawvere and Tierney over 25 years ago.
- It is are equivalent to Zermelo set theory with bounded comprehension and the axiom of choice.
- It is adequate to classical analysis.

The axioms use a two sorted language – a sort for objects and a sort for arrows. The axioms are partitioned into subsections: category axioms, finite completeness axioms, and topos axioms. The lists of primitives include informal explications. The actual axioms are numbered. The connections between the topos axioms and the IFF Foundation Ontology are indicated.

Category Axioms

To express the axioms for a category, we use the following terminology (primitives).

- For any object *A* the *identity* morphism on *A* is denoted here by '`Id(A)`'. In the Foundation Ontology a *class* (an object in the quasi-category of classes and functions) is declared by the '`(SET$class ?a)`' expression [axiom `SET$1`], and the identity is represented by the '`(SET.FTN$identity ?a)`' expression [axiom `SET.FTN$12`].

- Any *morphism f* with *source* (domain) object *A* and *target* (codomain) object *B* is denoted by '`f:A→B`'. In the Foundation Ontology a function (a morphism in the quasi-category of classes and functions) is declared by the '`(SET.FTN$function ?f)`' expression [axiom `SET.FTN$1`], and the source and target functions are represented by the '`(SET.FTN$source ?f)`' and '`(SET.FTN$target ?f)`' expressions [axioms `SET.FTN$2` and `SET.FTN$3`].

- The *composition* of two composable morphisms '`f:A→B`' and '`g:B→C`' is denoted by '`f·g`' in diagrammatic order. In the Foundation Ontology the composition of two composable functions is represented by the '`(SET.FTN$composition ?f ?g)`' expression [axiom `SET.FTN$11`].

The axioms for a category are as follows.

1. Two morphisms are *composable* when the target of the first is identical to the source of the second '`f:A→B`' and '`g:B→C`'. A composable pair of morphisms is expressed by the following axiom.

   ```
   ∀(f,g) [ ∃(A,B,C) (f:A→B & g:B→C) ⇔ ∃(h)(f·g = h) ].
   ```

   In the Foundation Ontology this equivalence for the case of composable functions is expressed in the axiom group `SET.FTN$11`.

2. The following axiom expresses the law of *associativity* of morphism composition.

   ```
   ∀(f,g,h,k,i,j) [ (f·g = k & k·h = j & g·h = i) ⇒ f·i = j ].
   ```

   In the Foundation Ontology the associative law of function composition is expressed by the theorem below `SET.FTN$11`.

3. The following pair of axioms express the two *identity laws* for categorical composition.

   ```
   ∀(f,A,B) [ f:A→B ⇒ (Id(A)·f = f & f·Id(B) = f) ].
   ```

   In the Foundation Ontology the identity laws for functions are expressed in the theorem below `SET.FTN$12`.

Finite Completeness Axioms

The terminology and axioms in this section extend those of the previous section to give a category with terminal object and finite products. For full finite completeness this relies upon the further topos axioms, which together with these axioms imply finite completeness and cocompleteness. McLarty's goal was to present a minimal set of axioms for a topos. This differs from the goals for the Foundation Ontology and IFF in general, which aim for completeness and high expressiveness. In particular, it is very important for other IFF metalevel ontologies to have access to a completely expressive terminology for pullbacks, since these are very heavily used.

- The terminal object is denoted by '`1`'. In the Foundation Ontology any singleton class is terminal. To be specific, the Foundation Ontology uses the unit class $1 = \{0\}$ as the terminal class. In the Foundation Ontology the terminal class (in the quasi-category of classes and functions) is declared by the '`SET.LIM$terminal`' and the '`SET.LIM$unit`' expressions [axiom `SET.LIM$1`].
- The binary Cartesian product for both objects and morphisms is denoted by '`_×_`'; in particular, for any two objects A_1 and A_2 the binary Cartesian product is denoted by '`A1×A2`'. In the Foundation Ontology the binary product CNG function for classes is represented by the term '`SET.LIM.PRD$binary-product`' [axiom `SET.LIM.PRD$6`], and the binary product CNG function for SET functions is represented by the term '`SET.LIM.PRD.FTN$binary-product`' [axiom `SET.LIM.PRD.FTN$11`].
- The two binary product projection morphisms are denoted by '`p1(_,_)`' and '`p2(_,_)`'. In the Foundation Ontology the two CNG binary product projection functions are represented by the terms '`SET.LIM.PRD$binary-product`'

The axioms for a category with terminal object and finite products are as follows.

4. An object 1 is *terminal* in a category when for any other object A there is a unique morphism $1_A : A \to 1$ from the object to the terminal object. The existence of a terminal object is stated by the following axiom.

   ```
   ∀(A) [ ∃!(f)(f:A→1) ].
   ```

   In the Foundation Ontology this universal property is expressed by the definition of the *unique* function '`SET.LIM$unique`' in axiom `SET.LIM$2`.

5. The source and target typing for the two *binary product projection functions* is declared by the following axioms.

   ```
   ∀(A,B) [ p1(A,B):A×B→A & p2(A,B):A×B→B ].
   ```

   In the Foundation Ontology declarations are is expressed by the binary Cartesian product projection axioms `SET.LIM.PRD$12` and `SET.LIM.PRD$13`.

6. The universality for the binary product is asserted by the following axiom.

   ```
   ∀(f,g,A,B,C) [ (f:C→A & g:C→B) ⇒
       ∃!(u)(u:C→A×B & u·p1(A,B) = f & u·p2(A,B) = g) ].
   ```

   In the Foundation Ontology this universal property is expressed by the definition of the *mediator* function '`SET.LIM.PRD$mediator`' in axiom `SET.LIM.PRD$14`.

7. The following axiom extends the binary product to functions.

   ```
   ∀(f,g,A,B,C,D) [ (f:A→B & g:C→D) ⇒
          ( (f×g):A×C→B×D
            & (f×g)·p1(B,D) = p1(A,C)·f
             & (fxg)·p2(B,D) = p2(A,C)·g ) ].
   ```

   In the Foundation Ontology this property is expressed by the definition of the function binary product '`SET.LIM.PRD.FTN$binary-product`' in axiom `SET.LIM.PRD.FTN$6`.

Topos Axioms

The terminology and axioms in this section extend those of the previous sections to a non-trivial Boolean topos.

- A *monomorphism* in a category corresponds to an injection. The assertion that a morphism is a monomorphism is denoted by '`mono(f)`'. An *epimorphism* in a category corresponds to a surjection. The assertion that a morphism is an epimorphism is denoted by '`epi(f)`'. In the setting of topos theory, monomorphisms are regarded as subobjects. In the Foundation Ontology the injection class is declared by the term '`SET.FTN$injection`' [axiom `SET.FTN$13`] and monomorphism class is declared by the term '`SET.FTN$monomorphism`' [axiom `SET.FTN$14`]; also, the surjection class is declared by the term '`SET.FTN$surjection`' [axiom `SET.FTN$15`] and epimorphism class is declared by the term '`SET.FTN$epimorphism`' [axiom `SET.FTN$16`].
- Given two objects A and B in a category the *exponent* '`B^A`' is the collection of all morphisms from A to B. In the Foundation Ontology the exponent class is declared by the term '`SET.TOP$exponent`' [axiom `SET.TOP$1`].
- Given two objects A and B in a Cartesian-closed category the *evaluation* morphism '`ev(A,B)`' evaluates morphisms: when applied to a morphism f from A to B and a value a in A it returns the image $f(a)$. In the Foundation Ontology the evaluation function is declared by the term '`SET.TOP$evaluation`' [axiom `SET.TOP$2`].
- The *truth object* is denoted by '`1+1`', a disjoint union of two copies of 1. In the Foundation Ontology the truth class is defined by $2 = \{0, 1\}$, where the elements are regarded as the truth values $0 = false$ and $1 = true$. It is isomorphic to the disjoint union $2 \cong 1+1$. In the Foundation Ontology the truth class is declared by the term '`SET.TOP$truth`' [axiom `SET.TOP$5`].
- The *binary coproduct injections* $in_1 : 1 \to 1+1$ and $in_2 : 1 \to 1+1$ are (function) elements of $1+1$ corresponding to the truth values *false* and *true*, respectively. In the Foundation Ontology the true function (second injection) is declared by the term '`SET.TOP$true`' [axiom `SET.TOP$6`].

The axioms for a Boolean topos are as follows.

8. A morphism is a *monomorphism* when it can be cancelled on the right (in diagrammatic order). Dually, a morphism is an *epimorphism* when it can be cancelled on the left. The definitions of monomorphism and epimorphisms are given by the following axioms.

   ```
   ∀(f) [ mono(f) ⇔ ∀(g,h)(g·f = h·f ⇒ g = h) ].
   ∀(f) [ epi(f) ⇔ ∀(g,h)(f·g = f·h ⇒ g = h) ].
   ```

   In the Foundation Ontology the first definition is expressed as the definition of the monomorphism class '`SET.FTN$monomorphism`' in axiom `SET.FTN$14`, and the second definition is expressed as the definition of the epimorphism class '`SET.FTN$epimorphism`' in axiom `SET.FTN$16`.

9. A *Cartesian-closed category* is a finitely-closed category whose binary product functor (needs a specific binary product here) is left adjoint to the exponent functor. This is expressed in the following axiom.

   ```
   ∀(f,A,B,C) [ f:CxA→B ⇒ ∃!(u)(u:C→B^A & (u×Id(A))·ev(A,B) = f) ].
   ```

   In the Foundation Ontology the "right adjoint" map, that takes the function f and returns the function u, is expressed as the definition of the *adjoint* function '`SET.TOP$adjoint`' in axiom `SET.TOP$3`.

10. An *elementary topos* is a Cartesian-closed category that has a subobject classifier – an object Ω and a morphism $true : 1 \to \Omega$ that satisfy the axiom below. A *classical topos* can use the Boolean truth object as subobject classifier $1+1 = \Omega$. The following subobject classifier axiom states that every subobject f of an object B has a unique characteristic function u of that object – a function that makes the Diagram 1 a pullback diagram.

    ```
    ∀(f,A,B) [ (f:A→B & mono(f)) ⇒ ∃!(u)(u:B→1+1 &
        ∀(h,k)((k·i2 = h·u) ⇒ ∃!(v)(v·f = h)) ) ].
    ```

In the Foundation Ontology the map, that takes the subobject *f* and returns the characteristic morphism *u*, is expressed as the definition of the *character* function '`SET.TOP$character`' in axiom `SET.TOP$7`.

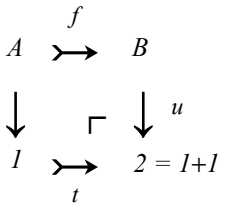


Diagram 1: Subobject Classifier

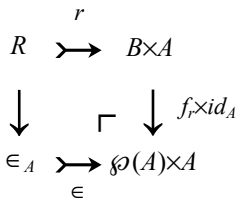


Diagram 2: Power Objects

11. A topos is *Boolean* when the truth injections $in_1 : 1 \to 1+1$ and $in_2 : 1 \to 1+1$ (truth value elements *false* and *true*) are *complements*. Equivalently, a topos is Boolean when the collection of subobjects of any object forms a Boolean algebra.

```
¬(i1 = i2)
& ∀(f,g,A) [ (f:1→A & g:1→A) ⇒ ∃!(u)(u:(1+1)→A & i1·u = f & i2·u = g) ].
```

In the Foundation Ontology the fact that the quasi-topos of classes and function is Boolean follows from the fact that the '`(el2ftn ?c)`' is bijective for each class *C* [axiom `SET.TOP$7`].

Classical Analysis

The following axioms allow us to get classical analysis by using natural numbers in a well-pointed topos with choice.

12. A *natural numbers object* in a category with terminal object and binary coproducts is an *initial algebra* for the endofunctor $T(-) = 1+(-)$ on the category. The following axiom encodes this idea.

```
∃(N,0,s) [ (0:1→N & s:N→N) &
     ∀(A,x,f) [ (x:1→A & f:A→A) ⇒ ∃!(u)(u:N→A & 0·u = x & s·u = u·f) ] ].
```

In the Foundation Ontology the natural numbers class '`SET.TOP$natural-numbers`', zero element '`SET.TOP$zero`' and successor endofunction '`SET.TOP$successor`' satisfy the axiom for a natural numbers object in the quasi-topos of classes and functions [axiom `SET.TOP$8`].

13. The following axiom is the *extensionality principle* for morphisms of a category with *1*. It states that *1* is a generator; that is, that morphisms are determined by their effect on the source (domain) elements. A category is *degenerate* when all of its objects are isomorphic. A non-degenerate topos that satisfies extensionality for morphisms is called *well-pointed*.

```
∀(f,g,A,B) [ (f:A→B & g:A→B) ⇒
    ∀(h) ( (h:1→A & (h·f = h·g)) ⇒ (f = g) ) ].
```

In the Foundation Ontology axiom `SET.TOP$9` states that functions (morphisms in the quasi-category of classes) satisfy the extensionality principle.

14. The following axiom is one variant of the *axiom of choice* – it uses the standard definition of an epi-morphism. The axiom of choice implies Boolean-ness for any topos.

```
∀(f,A,B) [ epi(f) ⇒ ∃(g)(g:B→A & g·f = Id(B)) ].
```

In the Foundation Ontology axiom `SET.TOP$10` states that the quasi-category of classes and functions satisfies the axiom of choice.

Sketches

Here is the paraphrase of a discussion of sketches by Vaughan Pratt.

> *A sketch is the categorical counterpart of a first-order theory. It specifies the language of the theory in terms of limits and colimits of diagrams. The language of (finitary) quantifier-free logic is representable entirely with finite product (FP) sketches, i.e. no colimits and only discrete limits. Finite limit (FL) sketches allow all limits, e.g. pullbacks which come in handy if you want to axiomatize composition of morphisms as a total operation (not possible with ordinary first order logic or FP sketches). Colimits extend the expressive power of sketches in much the same way that least-fixpoint operators extend the expressive power of first order logic (made precise by a very nice theorem of Adamek and Rosicky), but completely dually to limits. (Fixpoint operators are not obviously dual to anything in first order logic.) The machinery of sketches is either appealingly economical and elegant or repulsively complex and daunting depending on whether you look at it from the perspective of category theory or set theory. As a formalism for categorical foundations sketches have the same weakness as Colin McLarty's axiomatization of categories: they are based on ordinary categories, with no 2-cells. (Again let me stress the importance of 2-categories, i.e. not just line segments but surface patches, for foundations.) On the one hand I'm sure this is not an intrinsic limitation of sketches, on the other I don't know what's been done along those lines to date. Higher-dimensional sketches are surely well worth pursuing.*

It would be beneficial to develop sketches of the IFF metalevel for comparison and contrast.

Fibrations

There is a need to incorporate some aspect of fibrations and indexed categories in the Foundation Ontology. For one example, the (small) classification namespace represents the category Classification, which is part of a fibered span (Figure 3). For another example, the model namespace is a fibered span along instances and types; the type fiber is needed to specify satisfaction. See McLarty's suggestion to use Benabou's theory of fibrations and definability. My current belief is that fibers can be represented in the Foundation Ontology, without a full-blown representation of fibrations a la Benabou. However, this will be tested by further development of the category theory aspect of the Foundation Ontology, which needs to contain a theory of fibrations and indexed categories.

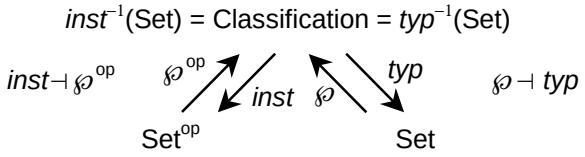


Figure 3: The Classification Fibered Span

Part I: The Large Aspect

The Namespace of Conglomerates

In a set-theoretic sense, this namespace sits at the top of the IFF Foundation Ontology. The suggested prefix for this namespace is '`CNG`', standing for conglomerates. When used in an external namespace, all terms that originate from this namespace can be prefixed with '`CNG`'. This namespace represents conglomerates, and their functions and relations. No sub-namespaces are needed. As illustrated in Diagram 1, conglomerates characterized the overall architecture for the large aspect of the Foundation Ontology. Nodes in this diagram represent conglomerates and arrows represent conglomerate functions. The small oval on the right, containing the function and class conglomerates, represents the namespace ('`SET`') of large sets (classes) and their functions. The next large oval, containing the conglomerates of Graphs and their Morphisms, represents the large graph namespace ('`GPH`'). Also indicated are namespaces for categories, functors, natural transformations and adjunctions.

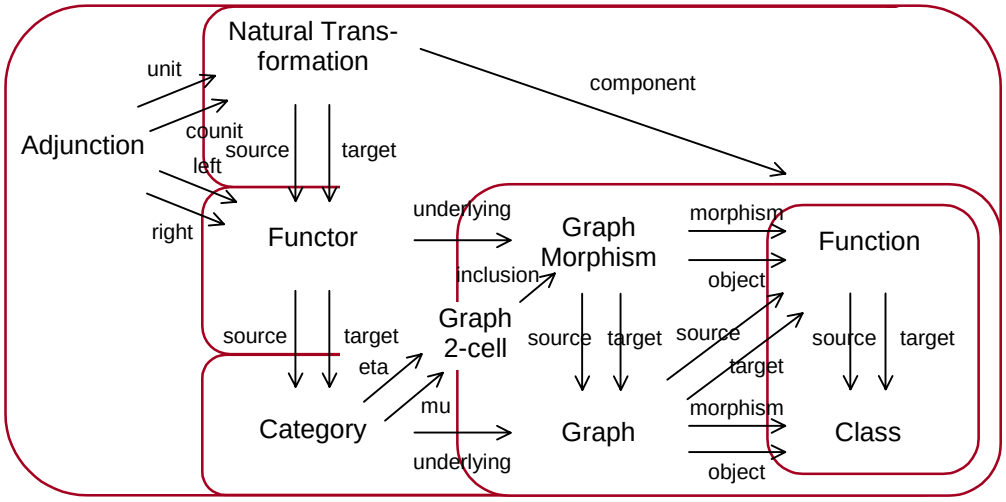


Diagram 1: Core Conglomerates and Functions

Conglomerates

`CNG`

The largest collection in the IFF Foundation Ontology is the *Conglomerate* collection. Conglomerates are collections of classes or individuals. In this version of the Foundation Ontology we will not need to axiomatize conglomerates in great detail. In addition to the conglomerate collection itself, we also provide simple terminology for conglomerate functions, conglomerate relations, and their signatures.

- Let '`conglomerate`' be the Foundation Ontology term that denotes the *Conglomerate* collection. Conglomerates are used at the core of the Foundation Ontology for several things: to specify the collection of classes, to specify the collection of class functions and their injection, surjection and bijection subcollections, and to specify the shape diagrams and cones for the various kinds of finite limits. Every conglomerate is represented as a KIF class. The collection of all conglomerates is not a conglomerate.

```
(1) (KIF$class collection)
    (forall (?c (collection ?c)) (KIF$class ?c))

(2) (collection conglomerate)
    (forall (?c (conglomerate ?c)) (collection ?c))
    (not (conglomerate conglomerate))
```

- There is a *subconglomerate* binary KIF relation.

```
(3) (KIF$relation subconglomerate)
```

```
(KIF$signature subconglomerate conglomerate conglomerate)
(forall (?k1 (conglomerate ?k1) ?k2 (conglomerate ?k2))
    (<=> (subconglomerate ?k1 ?k2) (KIF$subclass ?k1 ?k2)))
```

- There is a *disjoint* binary KIF relation.

```
(4) (KIF$relation disjoint)
    (KIF$signature disjoint conglomerate conglomerate)
    (forall (?c1 (conglomerate ?c1 ?c2 (conglomerate ?c2))
        (<=> (disjoint ?c1? ?c2)
             (not (exists (?x (?c1 ?x) (?c2 ?x))))))
```

- Let '`function`' be the Foundation Ontology term that denotes the *Function* collection. We assume the following definitional axiom has been stated in the Basic KIF Ontology.

```
(forall (?f)
    (<=> (KIF$function ?f)
         (and (KIF$relation ?f) (KIF$functional ?f))))
```

  - Every conglomerate function is represented as a KIF function. The signature of a conglomerate function is the same as its KIF signature, except that the KIF classes in the signature are conglomerates.

```
(4) (KIF$relation signature)

(5) (collection function)
    (forall (?f (function ?f)) (KIF$function ?f))

    (forall (?f (function ?f) @cng)
        (<=> (signature ?f @cng)
             (and (KIF$signature ?f @cng)
                  (function ?f)
                  (forall (?n (KIF$posint ?n) (=< ?n (length [@cng])))
                      (conglomerate ([@cng] ?n))))))
```

  - Let '`relation`' be the Foundation Ontology term that denotes the *Binary Relation* collection. Every conglomerate relation is represented as a binary KIF relation. The KIF signature of a conglomerate relation is given by its conglomerates.

```
(6) (collection relation)
    (forall (?r (relation ?r)) (and (KIF$relation ?r) (KIF$binary ?r)))

(7) (KIF$function conglomerate1)
    (KIF$signature conglomerate1 relation conglomerate)

(8) (KIF$function conglomerate2)
    (KIF$signature conglomerate2 relation conglomerate)

    (forall (?r (relation ?r))
        (KIF$signature ?r (conglomerate1 ?r) (conglomerate2 ?r)))

(9) (KIF$function extent)
    (KIF$signature extent relation conglomerate)

    (forall (?r (relation ?r) ?z ((extent ?r) ?z))
        (and (KIF$pair ?z)
             ((conglomerate2 ?r) (?z 1))
             ((conglomerate2 ?r) (?z 2))))

    (forall (?r (relation ?r)
             ?x1 ((conglomerate1 ?r) ?x1)
             ?x2 ((conglomerate2 ?r) ?x2))
         (<=> ((extent ?r) [?x1 ?x2])
              (?r ?x1 ?x2)))

    (forall (?r (relation ?r)
             ?s (relation ?s))
        (=> (and (= (conglomerate1 ?r) (conglomerate1 ?s))
                 (= (conglomerate2 ?r) (conglomerate2 ?s))
                 (= (extent ?r) (extent ?s)))
            (= r s)))
```

- There is a *subrelation* binary CNG relation that restricts the KIF subrelation relation to conglomerates.

```
(10) (KIF$relation subrelation)
     (KIF$signature subrelation relation relation)
     (forall (?r1 (relation ?r1) ?r2 (relation ?r2))
         (<=> (subrelation ?r1 ?r2) (KIF$subrelation ?r1 ?r2)))
```

The Namespace of Classes (Large Sets)

This is the core namespace in the Foundation Ontology. The suggested prefix for this namespace is 'SET', standing for large sets. When used in an external namespace, all terms that originate from this namespace can be prefixed with 'SET'. This namespace represents classes (large sets) and their functions. The terms listed in Table 1 are declared and axiomatized in this namespace. As indicated in the left-hand column of Table 1, several sub-namespaces are needed.

Table 1: Terms introduced in the core namespace

	CNG$conglomerate	Unary CNG$function	Binary CNG$function
SET	'class'		
SET .FTN	'function' 'parallel-pair' 'injection', 'surjection', 'bijection' 'monomorphism', 'epimorphism', 'isomorphism'	'source', 'target', 'identity' 'image', 'inclusion', 'fiber', 'inverse-image'	'composition'
SET .LIM	'unit', 'terminal'	'unique' 'tau-cone', 'tau'	
SET .LIM .PRD	'diagram', 'pair' 'cone'	'class1', 'class2', 'opposite' 'cone-diagram', 'vertex', 'first', 'second' 'limiting-cone', 'limit', 'binary-product', 'projection1, 'projection2' 'mediator' 'binary-product-opspan' 'tau-cone', 'tau'	'pairing-cone', 'pairing'
SET .LIM .PRD .FTN	'pair'	'source', 'target', 'class1', 'class2' 'binary-product'	
SET .LIM .EQU	'diagram', 'parallel-pair', 'cone'	'source', 'target', 'function1', 'function2' 'cone-diagram', 'vertex', 'function' 'limiting-cone', 'limit', 'equalizer', 'canon' 'mediator' 'kernel-diagram', 'kernel'	
SET .LIM .SEQU	'lax-diagram', 'lax-parallel-pair', 'lax-cone'	'order', 'source', 'function1', 'function2', 'parallel-pair' 'lax-cone-diagram', 'vertex', 'function' 'limiting-lax-cone', 'lax-limit', 'subequalizer', 'subcanon' 'mediator'	

SET .LIM .PBK	'diagram', 'opspan', 'cone'	'opvertex', 'opfirst', 'opsecond', 'opposite' 'pair' 'cone-diagram', 'vertex', 'first', 'second' 'limiting-cone', 'limit', 'pullback', 'projection1', 'projection2', 'relation' 'mediator' 'fiber', 'fiber1', 'fiber2', 'fiber12', 'fiber21' 'fiber-embedding', 'fiber1-embedding', 'fiber2-embedding', 'fiber12-embedding', 'fiber21-embedding' 'fiber1-projection', 'fiber2-projection' 'kernel-pair-diagram', 'kernel-pair' 'tau-cone', 'tau'	'pairing-cone', 'pairing'
SET .TOP		'evaluation' 'adjoint' 'subclass' 'element', 'el2ftn' 'truth', 'true' 'character'	'exponent' 'constant'

The signatures for some of the relations and functions in the core namespace are listed in Table 2.

Table 2: Signatures for some relations and functions in the conglomerate and core namespaces

subconglomerate $\subseteq$ *conglomerate* $\times$ *conglomerate* *signature* $\subseteq$ *function* $\times$ *KIF$sequence* *subclass* $\subseteq$ *class* $\times$ *class* *disjoint* $\subseteq$ *class* $\times$ *class* *partition* $\subseteq$ *class* $\times$ *KIF$sequence* *restriction* $\subseteq$ *function* $\times$ *CNG$function* *restriction-pullback* $\subseteq$ *function* $\times$ *CNG$function*	*source, target* : *function* $\rightarrow$ *class* *identity, range* : *function* $\rightarrow$ *class* *vertex* : *span* $\rightarrow$ *class* *first*, *second* : *span* $\rightarrow$ *function* *opvertex* : *opspan* $\rightarrow$ *class* *opfirst*, *opsecond* : *opspan* $\rightarrow$ *function* *opposite* : *opspan* $\rightarrow$ *opspan*	*composition* : *function* $\times$ *function* $\rightarrow$ *function*
	unique : *class* $\rightarrow$ *function* *cone-opspan* : *cone* $\rightarrow$ *opspan* *vertex* : *cone* $\rightarrow$ *class* *first*, *second*, *mediator* : *cone* $\rightarrow$ *function* *limiting-cone* : *opspan* $\rightarrow$ *cone*	*binary-product* : *class* $\times$ *class* $\rightarrow$ *class* *binary-product-opspan* : *class* $\times$ *class* $\rightarrow$ *opspan*
	power : *class* $\rightarrow$ *relation*	*exponent* : *class* $\times$ *class* $\rightarrow$ *class* *evaluation* : *class* $\times$ *class* $\rightarrow$ *function*

Classes

SET

The collection of all classes is denoted by *Class*. It is an example of a *conglomerate* in (Adámek, Herrlich & Strecker 1990). Also, since we need power classes, no universal class *Thing* is postulated (We may want to postulate the existence of a universal conglomerate instead).

- Let 'class' be the SET namespace term that denotes the *Class* collection. Classes are mainly used in IFF to specify the object and morphism collections of large categories such as Classification. Semantically, every class is a conglomerate; hence syntactically, every class is represented as a KIF class. The collection of all classes is not a class.

```
(1) (CNG$conglomerate class)
    (forall (?c (class ?c)) (CNG$conglomerate ?c))
    (not (class class))
```

- There is a *subcollection* binary KIF relation that compares a class to a conglomerate by restricting the subconglomerate relation. There is a *subclass* binary KIF relation that restricts the subconglomerate relation to classes.

```
(2) (CNG$relation subcollection)
    (CNG$signature subcollection class CNG$conglomerate)
    (forall (?c1 (class ?c1) ?c2 (CNG$conglomerate ?c2))
        (<=> (subcollection ?c1 ?c2) (CNG$subconglomerate ?c1 ?c2)))

    (CNG$relation subclass)
    (CNG$signature subclass class class)
    (forall (?c1 (class ?c1) ?c2 (class ?c2))
        (<=> (subclass ?c1 ?c2) (CNG$subconglomerate ?c1 ?c2)))
```

- There is a *disjoint* binary CNG relation.

```
(3) (CNG$relation disjoint)
    (KIF$signature disjoint class class)
    (forall (?c1 (class ?c1 ?c2(class ?c2)
        (<=> (disjoint ?c1? ?c2)
             (not (exists (?x (?c1 ?x) (?c2 ?x))))))
```

- Any SET class can be partitioned. A partition of the class C by the sequence of classes $C_1, \ldots, C_n$ is denoted by the expression '`(partition ?c [?c1 … ?cn])`'. All elements in a partition are classes.

```
(4) (CNG$relation partition)
    (CNG$signature partition class KIF$sequence)
    (forall (?c (class ?c) ?p (KIF$sequence ?p))
        (=> (partition ?c ?p)
            (and (forall (?pi (KIF$element-of ?pi ?p))
                     (and (class ?pi) (subclass ?pi ?c)))
                 (forall (?j (=< ?j (length ?p)) ?k (=< ?k (length ?p)))
                     (=> (not (= ?i ?j)) (disjoint (?s ?j) (?s ?k)))))))
```

- For any sequence of SET classes there is a union class and an intersection class.

```
(5) (CNG$function union)
    (forall (@cng ?x)
        (<=> ((union @cng) ?x)
             (exists (?n (KIF$posint ?n) (=< ?n (length [@cng])))
                 (([@cng] ?n) ?x))))

(6) (CNG$function intersection)
    (forall (@cng ?x)
        (<=> ((intersection @cng) ?x)
             (forall (?n (KIF$posint ?n) (=< ?n (length [@cng])))
                 (([@cng] ?n) ?x))))
```

- There is a foundational question here: "Is the power of a class another class?" We have taken the strong answer "Yes!" and made the power of a class a class. The motivation is the need to define fibers. More strongly, we are assuming that classes and their functions satisfy the axioms of a topos. Eventually we may need to use Jean Benabou's foundational approach here: see "Fibered categories and the foundations of naive category theory" by Jean Benabou, in the *Journal of Symbolic Logic* 50, 10–37, 1985. But for now we only define the fibrational structure that seems to be required. For any class X the *power-class* over X is the collection of all subclasses of X. There is a unary CNG '`power`' function that maps a class to its associated power.

```
(7) (CNG$function power)
    (CNG$signature power SET$class SET$class)
    (forall (?c1 (SET$class ?c1) ?c0)
        (<=> ((power ?c1) ?c0) (SET$subclass ?c0 ?c1)))
```

Functions

SET.FTN

A (class) function (Figure 1) is a special case of a unary conglomerate function with source and target classes. A class function is also known as a SET function. An SET function is intended to be an abstract semantic notion. Syntactically however, every function is represented as a unary KIF function. The signature of SET functions, considered to be CNG functions, is given by their source and target. A SET function with *source* (domain) class X and *target* (codomain) class Y is a triple (X, Y, f), where the class $f \subseteq X \times Y$ is the underlying *relation* of the function. We use the notation $f : X \to Y$ to indicate the source-target typing of a class function. All SET functions are total, hence must satisfy the constraint that for every $x \in X$ there is a unique $y \in Y$ with $(x, y) \in f$. We use the notation $f(x) = y$ for this instance.

$$X \xrightarrow{f} Y$$

Figure 1: Class Function

For SET functions both composition and identities are defined. Given two functions $f : X \to Y$ and $g : Y \to Z$ the *composition* function $f \cdot g : X \to Z$ is defined by $f \cdot g\ (x) = g(f(x))$ for all $x \in X$. Composition is associative: $f \cdot (g \cdot h) = (f \cdot g) \cdot h$. For any class X there is an identity function $id_X : X \to X$. Identity satisfies the identity laws: $id_X \cdot f = f = f \cdot id_Y$. Composition and identity make the collections of classes and functions into a quasi-category. This is not a true category, since the collection of all classes and the collection of all class functions are not classes, but conglomerates.

- Let '`function`' be the SET namespace term that denotes the *Function* collection.

```
(1) (CNG$conglomerate function)
    (forall (?f (function ?f)) (CNG$function ?f))

(2) (CNG$function source)
    (CNG$signature source function SET$class)

(3) (CNG$function target)
    (CNG$signature target function SET$class)

    (forall (?f (function ?f))
        (CNG$signature ?f (source ?f) (target ?f)))

    (forall (?f (function ?f))
        (forall (?x ((source ?f) ?x))
            (exists (?y ((target ?f) ?y))
                (= (?f ?x) ?y))))
```

  - Any function can be embedded as a binary relation.

```
(4) (CNG$function fn2rel)
    (CNG$signature fn2rel function REL$relation)
    (forall (?f (function ?f)
        (and (= (REL$class1 (fn2rel ?f)) (source ?f))
             (= (REL$class2 (fn2rel ?f)) (target ?f))))
    (forall (?f (function ?f)
             ?x ((source ?f) ?x)
             ?y ((target ?f) ?y))
        (<=> ((REL$extent (fn2rel ?f)) [?x ?y])
             (= (?f ?x) ?y)))
```

- A class function $f : C \to D$ is an *ordinary restriction* of a conglomerate function $F : \check{C} \to \check{D}$ when the source (target) of f is a subcollection of the source (target) of F and the functions agree (on source elements of f); that is, the functions commute (Diagram 1) with the source/target inclusions. Ordinary restriction is a constraint on the conglomerate function – it says that on the class function source subcollection the conglomerate function maps into the class function target subcollection.

```
(5) (KIF$relation restriction)
    (KIF$signature restriction function CNG$function)
    (forall (?ftn ?FTN (function ?ftn) (CNG$function ?FTN))
        (<=> (restriction ?ftn ?FTN)
             (exists (?cng1 (CNG$conglomerate ?cng1)
                      ?cng2 (CNG$conglomerate ?cng2))
                 (and (CNG$signature ?FTN ?cng1 ?cng2)
                      (CNG$subconglomerate (source ?ftn) ?cng1)
                      (CNG$subconglomerate (target ?ftn) ?cng2)
                      (forall (?x ((source ?ftn) ?x))
                          (= (?ftn ?x) (?FTN ?x))))))
```

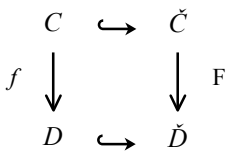


Diagram 1: Ordinary restriction

- When the source class is conceptually binary by being the pullback of some opspan, the restriction operator is more complicated. The pullback restriction operator is defined as follows. A (conceptually binary) class function f is a *pullback restriction* of a binary conglomerate function F when
  1. there is a class opspan $f_1 : C_1 \to C, f_2 : C_2 \to C$ with pullback $1^{st} : C_1 \times_C C_2 \to C_1$, $2^{nd} : C_1 \times_C C_2 \to C_2$
  2. the source and target typings are $f : C_1 \times_C C_2 \to C_3$ and $F : K_1 \times K_2 \to K_3$, where C_n is a subconglomerate of K_n for $n = 1, 2, 3$
  3. there are conglomerate functions $F_1 : K_1 \to K$, $F_2 : K_2 \to K$, where f_n is a restriction of F_n, $n = 1, 2$
  4. the domain of F is the conceptual pullback:
     $\forall x_1 \in K_1$ and $x_2 \in K_2$, $\exists y \in K$ such that $F(x_1, x_2) = y$ iff $F_1(x_1) = F_2(x_2)$
  5. class pullback constraints equal set pullback constraints on sets:
     $\forall\ x_1 \in\ C_1$ and $x_2 \in\ C_2$, $[x_1, x_2] \in\ C_1 \times_C C_2$ iff $F_1(x_1) = F_2(x_2)$
  6. f and F agree on the pullback:
     $\forall [x_1, x_2] \in\ C_1 \times_C C_2, f([x_1, x_2]) = F(x_1, x_2)$.

  Pullback restriction is also a constraint on the conglomerate function – it says that on the class function source subcollection the conglomerate function maps into the class function target subcollection. The special case of binary product restriction is included in binary pullback restriction.

```
(6) (KIF$relation restriction-pullback)
    (KIF$signature restriction-pullback function CNG$function)
    (forall (?ftn (function ?ftn)
             ?FTN (CNG$function ?FTN))
        (<=> (restriction-pullback ?f ?FTN)
             (exists (?src-opspan (SET.LIM.PBK$opspan ?src-opspan)
                      ?cng1 (CNG$conglomerate ?cng1)
                      ?cng2 (CNG$conglomerate ?cng2)
                      ?cng3 (CNG$conglomerate ?cng3)
                      ?src1 (SET$class ?src1)
                      ?src2 (SET$class ?src2)
                      ?tgt (SET$class ?tgt))
                      ?FTN1 (CNG$function ?FTN1)
                      ?FTN2 (CNG$function ?FTN2)
                 (and (CNG$signature ?FTN ?cng1 ?cng2 ?cng3)
                      (= (SET.FTN$source ?f) (SET.LIM.PBK$pullback ?src-opspan))
                      (= (SET.FTN$target ?ftn) ?tgt)
                      (= ?src1 (SET.FTN$source (SET.LIM.PBK$opfirst ?src-opspan)))
                      (= ?src2 (SET.FTN$source (SET.LIM.PBK$opsecond ?src-opspan)))
                      (SET$subclass ?src1 ?cng1)
                      (SET$subclass ?src2 ?cng2)
                      (SET$subclass ?tgt ?cng3)
                      (CNG$signature ?FTN1 ?cng1 ?cng3)
                      (CNG$signature ?FTN2 ?cng2 ?cng3)
                      (restriction (SET.LIM.PBK$opfirst ?src-opspan) ?FTN1)
                      (restriction (SET.LIM.PBK$opsecond ?src-opspan) ?FTN2)
                      (forall (?x1 (?cng1 ?x1) ?x2 (?cng2 ?x2))
                            (<=> (exists (?y (?cng3 ?y) (= (?g ?x1 ?x2) ?y))
                                     (= (?FTN1 ?x1) (?FTN2 ?x2))))
                      (forall (?x1 (?src1 ?x1) ?x2 (?src2 ?x2))
                          (and (<=> ((SET.FTN$source ?f) [?x1 ?x2])
                                    (exists (?y (?cng3 ?y) (= (?g ?x1 ?x2) ?y)))
                               (= (?ftn [?x1 ?x2]) (?FTN ?x1 ?x2)))))))))
```

- The binary *subequalizer restriction* is defined in a similar manner. Subequalizer restriction is also a constraint on the conglomerate function – it says that on the class function source subcollection the conglomerate function maps into the class function target subcollection. The special case of binary *equalizer restriction* is included in binary subequalizer restriction.

```
(7) (KIF$relation restriction-subequalizer)
```

…

- An *endofunction* is a function on a particular class; that is, it has that class as both source and target.

```
(8) (CNG$conglomerate endofunction)
    (forall (?f (endofunction ?f))
        (and (function ?f)
             (= (source ?f) (target ?f))))
```

- For any subclass relationship $A \subseteq B$ there is a unary CNG *inclusion* function $\subseteq_{A,B} : A \to B$.

```
(9) (CNG$function inclusion)
    (CNG$signature inclusion class class function)
    (forall (?a (class ?a) ?b (class ?b))
        (and (= (source (inclusion ?a ?b)) ?a)
             (= (target (inclusion ?a ?b)) ?b)))
    (forall (?a (class ?a) ?b (class ?b)
             ?x (?a ?x))
        (= ((inclusion ?a ?b) ?x) ?x))
```

- There is a unary CNG *fiber* function. For any class function $f : A \to B$, and any element $y \in B$, the fiber of y along f is the class $f^{-1}(y) = \{x \in A \mid f(x) = y\} \subseteq A$. For convenience we define a special fiber inclusion function $\subseteq_{f,y} : f^{-1}(y) \to A$ for any element $y \in B$.

```
(10) (CNG$function fiber)
     (CNG$signature fiber function function)
     (forall (?f (function ?f))
         (and (= (source (fiber ?f)) (target ?f))
              (= (target (fiber ?f)) (SET$power (source ?f)))))
     (forall (?f (function ?f)
              ?y ((target ?f) ?y)
              ?x ((source ?f) ?x))
         (<=> (((fiber ?f) ?y) ?x)
              (= (?f ?x) ?y))))

(11) (CNG$fiber-inclusion)
     (CNG$signature fiber-inclusion function CNG$function)
     (forall (?f (function ?f))
         (CNG$signature (fiber-inclusion ?f) (target ?f) function))
     (forall (?f (function ?f)
              ?y ((target ?f) ?y))
         (and (= (source ((fiber-inclusion ?f) ?y)) ((fiber ?f) ?y))
              (= (target ((fiber-inclusion ?f) ?y)) (source ?f))
         (= ((fiber-inclusion ?f) ?y)
            (inclusion ((fiber ?f) ?y) (source ?f))))
```

- There is a unary CNG *inverse image* function. For any class function $f : A \to B$ there is an inverse image function $f^{-1} : \wp B \to \wp A$ defined by $f^{-1}(Y) = \{x \in A \mid f(x) \in Y\} \subseteq A$ for any subset $Y \subseteq B$.

```
(12) (CNG$function inverse-image)
     (CNG$signature inverse-image function function)
     (forall (?f (function ?f))
         (and (= (source (inverse-image ?f)) (SET$power (target ?f)))
              (= (target (inverse-image ?f)) (SET$power (source ?f)))))
     (forall (?f (function ?f)
              ?y ((SET$power (target ?f)) ?y)
              ?x ((source ?f) ?x))
         (<=> (((inverse-image ?f) ?y) ?x)
              (?y (?f ?x))))
```

- There is an binary CNG function *composition* that takes two composable SET functions and returns their composition.

```
(13) (CNG$function composition)
     (CNG$signature composition function function function)

     (forall (?f1 (function ?f1) ?f2 (function ?f2))
         (<=> (exists (?f) (= (composition ?f1 ?f2) ?f))
              (= (target ?f1) (source ?f2)))))

     (forall (?f1 (function ?f1) ?f2 (function ?f2))
         (=> (= (target ?f1) (source ?f2))
             (and (= (source (composition ?f1 ?f2)) (source ?f1))
                  (= (target (composition ?f1 ?f2)) (target ?f2)))))

     (forall (?f1 (function ?f1) ?f2 (function ?f2))
         (=> (= (target ?f1) (source ?f2))
             (forall (?x ((source ?f1) ?x) ?z ((target ?f2) ?z))
                 (<=> (= ((composition ?f1 ?f2) ?x) ?z)
                      (exists (?y ((target ?f1) ?y))
                          (and (= (?f1 ?x) ?y) (= (?f2 ?y) ?z)))))))
```

- Composition satisfies the usual *associative law*.

```
(forall (?f1 (function ?f1) ?f2 (function ?f2) ?f3 (function ?f3))
    (=> (and (= (target ?f1) (source ?f2))
             (= (target ?f2) (source ?f3)))
        (= (composition ?f1 (composition ?f2 ?f3))
           (composition (composition ?f1 ?f2) ?f3))))
```

- There is an unary CNG function *identity* that takes a class and returns its associated identity function.

```
(14) (CNG$function identity)
     (CNG$signature identity SET$class function)

     (forall (?c (SET$class ?c))
         (and (= (source (identity ?c)) ?c)
              (= (target (identity ?c)) ?c))))

     (forall (?c ?x ?y (SET$class ?c))
         (<=> (= ((identity ?c) ?x) ?y)
              (= ?x ?y)))
```

- The identity satisfies the usual *identity laws* with respect to composition.

```
(forall (?f (function ?f))
    (and (= (composition (identity (source ?f)) ?f) ?f)
         (= (composition ?f (identity (target ?f))) ?f)))
```

- The *parallel pair* is the equivalence relation on functions, where two functions are related when they have the same source and target classes.

```
(15) (REL.ENDO$equivalence-relation parallel-pair)
     (= (REL.ENDO$class parallel-pair) function)
     (forall (?f (function ?f) ?g (function ?g))
         (<=> ((REL.ENDO$extent parallel-pair) [?f ?g])
              (and (= (source ?f) (source ?g))
                   (= (target ?f) (target ?g))))))
```

- A function is an *injection* when no distinct source elements have the same image. A function is an *monomorphism* when right composition by the function is injective.

```
(16) (CNG$conglomerate injection)
     (CNG$subconglomerate injection function)
     (forall (?f (function ?f))
         (<=> (injection ?f)
              (forall (?x1 ((source ?f) ?x1)
                       ?x2 ((source ?f) ?x2))
                  (=> (= (?f ?x1) (?f ?x2))
                      (= ?x1 ?x2))))))

(17) (CNG$conglomerate monomorphism)
     (CNG$subconglomerate monomorphism function)
     (forall (?f (function ?f))
         (<=> (monomorphism ?f)
```

```
                (forall (?g1 (function ?g1)
                         ?g2 (function ?g2))
                    (=> (and (= (target ?g1) (source ?f))
                             (= (target ?g2) (source ?f))
                             (= (composition ?g1 ?f) (composition ?g2 ?f))
                        (= ?g1 ?g2)))))
```

- We can prove the theorem that a function is an injection exactly when it is a monomorphism.

```
(= injection monomorphism)
```

- A function is a *surjection* when all elements of the target class are images. A function is *epimorphism* when left composition by the function is injective.

```
(18) (CNG$conglomerate surjection)
     (CNG$subconglomerate surjection function)
     (forall (?f (function ?f))
         (<=> (surjection ?f)
              (forall (?y ((target ?f) ?y))
                  (exists (?x ((source ?f) ?x))
                      (= (?f ?x) ?y)))))

(19) (CNG$conglomerate epimorphism)
     (CNG$subconglomerate epimorphism function)
     (forall (?f (function ?f))
         (<=> (epimorphism ?f)
              (forall (?g1 (function ?g1)
                       ?g2 (function ?g2))
                  (=> (and (= (target ?f) (source ?g1))
                           (= (target ?f) (source ?g2))
                           (= (composition ?f ?g1) (composition ?f ?g2))
                      (= ?g1 ?g2)))))
```

- We can prove the theorem that a function is a surjection exactly when it is an epimorphism.

```
(= surjection epimorphism)
```

- A function is a *bijection* when it is both an injection and a surjection. A function is an *isomorphism* when it is both a monomorphism and an epimorphism.

```
(20) (CNG$conglomerate bijection)
     (CNG$subconglomerate bijection function)
     (forall (?f (function ?f))
         (<=> (bijection ?f)
              (and (injection ?f) (surjection ?f))))

(21) (CNG$conglomerate isomorphism)
     (CNG$subconglomerate isomorphism function)
     (forall (?f (function ?f))
         (<=> (isomorphism ?f)
              (and (monomorphism ?f) (epimorphism ?f))))
```

- We can prove the theorem that a function is a bijection exactly when it is an isomorphism.

```
(= bijection isomorphism)
```

- There is a unary CNG function *image* that denotes exactly the image class of the function.

```
(22) (CNG$function image)
     (CNG$signature image function SET$class)
     (forall (?f ?y (function ?f))
         (<=> ((image ?f) ?y)
              (exists (?x) (and ((source ?f) ?x) (= (?f ?x) ?y)))))
```

- For any two functions $f_1, f_2 : A \to \boldsymbol{B} = \langle B, \leq \rangle$ whose target is an order, f_1 is a *subfunction* of f_2 when the images are ordered.

```
(23) (CNG$relation subfunction)
     (CNG$signature subfunction function function ORD$order)
     (forall (?f1 (function ?f2)
              ?f2 (function ?f2)
              ?o (ORD$order ?o))
```

```
(<=> (subfunction ?f1 ?f2 ?o)
     (and (= (source ?f1) (source ?f2))
          (= (target ?f1) (target ?f2))
          (= (target ?f1) (ORD$class ?o))
          (forall (?x ((source ?f1) ?x))
              ((ORD$relation ?o) (?f1 ?x) (?f2 ?x))))))
```

Finite Completeness

SET.LIM

Here we present axioms that make the quasi-category of classes and functions finitely complete. We assert the existence of terminal classes, binary products, equalizers of parallel pairs of functions and pullbacks of opspans. All are defined to be specific classes – for example, the binary product is the Cartesian product. Because of commonality, the terminology for binary product, equalizer, subequalizer and pullbacks are put into sub-namespace. The *diagrams* and *limits* are denoted by both generic and specific terminology.

The Terminal Class

- There is a *terminal* (or *unit*) class *1*. This is specific, and contains exactly one member. For each class *C* there is a *unique* function $!_C : C \to !$ to the unit class. There is a unary CNG '`unique`' function that maps a class to its associated unique SET function. We use a KIF definite description to define the unique function.

```
(1) (SET$class unit)
    (SET$class terminal)
    (= terminal unit)
    (unit 0)
    (forall (?x (unit ?x)) (= ?x 0))

(2) (CNG$function unique)
    (CNG$signature unique SET$class function)
    (forall (?c (SET$class ?c))
        (= (unique ?c)
           (the (?f (SET.FTN$function ?f))
                (and (= (SET.FTN$source ?f) ?c)
                     (= (SET.FTN$target ?f) unit)
                     (forall (?x (?c ?x)) (= (?f ?x) 0)))))))
```

Binary Products

SET.LIM.PRD

A *binary product* is a finite limit for a diagram of shape • •. Such a diagram (of classes and functions) is called a *pair* of classes.

- A *pair* (of classes) is the appropriate base diagram for a binary product. Each pair consists of a pair of classes called *class1* and *class2*. Let either '`diagram`' or '`pair`' be the SET namespace term that denotes the *Pair* collection. Pairs are determined by their two component classes.

```
(1) (CNG$conglomerate diagram)
    (CNG$conglomerate pair)
    (= pair diagram)

(2) (CNG$function class1)
    (CNG$signature class1 diagram SET$class)

(3) (CNG$function class2)
    (CNG$signature class2 diagram SET$class)

    (forall (?p (diagram ?p) ?q (diagram ?q))
        (=> (and (= (class1 ?p) (class1 ?q))
                 (= (class2 ?p) (class2 ?q)))
            (= ?p ?q)))
```

- Every pair has an opposite.

```
(4) (CNG$function opposite)
    (CNG$signature opposite pair pair)
```

```
(forall (?p (pair ?p))
    (and (= (class1 (opposite ?p)) (class2 ?p))
         (= (class2 (opposite ?p)) (class1 ?p))))
```

- The opposite of the opposite is the original pair – the following theorem can be proven.

```
(forall (?p (pair ?p))
    (= (opposite (opposite ?p)) ?p))
```

- A *product cone* is the appropriate cone for a binary product. A product cone (Figure 2) consists of a pair of functions called *first* and *second*. These are required to have a common source class called the *vertex* of the cone. Each product cone is over a pair. A product cone is the very special case of a cone over a pair (of classes). Let '`cone`' be the SET term that denotes the *Product Cone* collection.

```
(5) (CNG$conglomerate cone)

(6) (CNG$function cone-diagram)
    (CNG$signature cone-diagram cone diagram)

(7) (CNG$function vertex)
    (CNG$signature vertex cone SET$class)

(8) (CNG$function first)
    (CNG$signature first cone SET.FTN$function)
    (forall (?r (cone ?r))
        (and (= (SET.FTN$source (first ?r)) (vertex ?r))
             (= (SET.FTN$target (first ?r)) (class1 (cone-diagram ?r)))))

(9) (CNG$function second)
    (CNG$signature second cone SET.FTN$function)
    (forall (?r (cone ?r))
        (and (= (SET.FTN$source (second ?r)) (vertex ?r))
             (= (SET.FTN$target (second ?r)) (class2 (cone-diagram ?r)))))
```

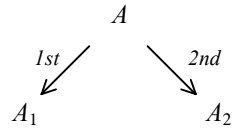


Figure 2: Product Cone

- There is a unary CNG function '`limiting-cone`' that maps a pair (of classes) to its binary product (limiting binary product cone) (Figure 3). Axiom (*) asserts that this function is total. This, along with the universality of the mediator function, implies that a binary product exists for any pair of classes. The vertex of the binary product cone is a specific *Binary Cartesian Product* class given by the CNG function '`binary-product`'. It comes equipped with two CNG projection functions '`projection1`' and '`projection2`'. This notation is for convenience of reference. It is used for pullbacks in general. Axiom (#) ensures that this product is specific – that it is exactly the Cartesian product of the pair of classes. Axiom (%) ensures that the projection functions are also specific.

```
(10) (CNG$function limiting-cone)
     (CNG$signature limiting-cone diagram cone)
 (*) (forall (?p (diagram ?p))
         (exists (?r (cone ?r))
             (= (limiting-cone ?p) ?r)))
     (forall (?p (diagram ?p))
         (= (cone-diagram (limiting-cone ?p)) ?p))

(11) (CNG$function limit)
     (CNG$function binary-product)
     (= binary-product limit)
     (CNG$signature limit diagram SET$class)
     (forall (?p (diagram ?p))
         (= (limit ?p) (vertex (limiting-cone ?p))))
 (#) (forall (?p (diagram ?p) ?z (KIF$pair ?z))
         (<=> ((limit ?p) ?z)
              (and ((class1 ?p) (?z 1))
                   ((class2 ?p) (?z 2)))))

(12) (CNG$function projection1)
     (CNG$signature projection1 diagram SET.FTN$function)
     (forall (?p (diagram ?p))
         (and (= (SET.FTN$source (projection1 ?p)) (limit ?p))
              (= (SET.FTN$target (projection1 ?p)) (class1 ?p))
              (= (projection1 ?p) (first (limiting-cone ?p)))))
```

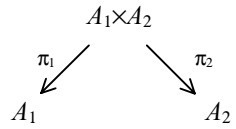


Figure 3: Limiting Cone

```
(13) (CNG$function projection2)
     (CNG$signature projection2 diagram SET.FTN$function)
     (forall (?p (diagram ?p))
         (and (= (SET.FTN$source (projection2 ?p)) (limit ?p))
              (= (SET.FTN$target (projection2 ?p)) (class2 ?p))
              (= (projection2 ?p) (second (limiting-cone ?p)))))

 (%) (forall (?p (diagram ?p) ?z ((limit ?p) ?z))
     (and (= ((projection1 ?p) ?z) (?z 1))
          (= ((projection2 ?p) ?z) (?z 2))))
```

- There is a *mediator* function from the vertex of a product cone over a pair (of classes) to the binary product of the pair. This is the unique function that commutes with first and second. We use a KIF definite description abbreviation to define this. Existence and uniqueness represents the universality of the binary product operator. We have also introduced a convenience term '`pairing`'. With a pair parameter, the binary CNG function '`(pairing ?p)`' maps a pair of SET functions, that form a binary cone with the class pair, to their mediator (pairing) function.

```
(14) (CNG$function mediator)
     (CNG$signature mediator cone SET.FTN$function)
     (forall (?r (cone ?r))
         (= (mediator ?r)
            (the (?f (SET.FTN$function ?f))
                 (and (= (SET.FTN$source ?f) (vertex ?r))
                      (= (SET.FTN$target ?f) (limit (cone-diagram ?r))))))
                      (= (SET.FTN$composition ?f (projection1 (cone-diagram ?r)))
                         (first ?r))
                      (= (SET.FTN$composition ?f (projection2 (cone-diagram ?r)))
                         (second ?r))))))

(15) (KIF$function pairing-cone)
     (KIF$signature pairing-cone diagram CNG$function)
     (forall (?p (diagram ?p))
         (and (CNG$signature (pairing-cone ?p)
               SET.FTN$function SET.FTN$function cone)
              (=> (exists (?f1 ?f2 (SET.FTN$function ?f1) (SET.FTN$function ?f2)
                          ?r (cone ?r))
                          (= (((pairing-cone ?p) [?f1 ?f2]) ?r))
                  (and (= (SET.FTN$source ?f1) (SET.FTN$source ?f2))
                       (= (SET.FTN$target ?f1) (class1 ?p))
                       (= (SET.FTN$target ?f2) (class2 ?p))))))
     (forall (?p (diagram ?p)
              ?f1 (SET.FTN$function ?f1) ?f2 (SET.FTN$function ?f2))
         (=> (and (= (SET.FTN$source ?f1) (SET.FTN$source ?f2))
                  (= (SET.FTN$target ?f1) (class1 ?p))
                  (= (SET.FTN$target ?f2) (class2 ?p)))
             (and (= (cone-diagram ((pairing-cone ?p) ?f1 ?f2)) ?p)
                  (= (vertex ((pairing-cone ?p) ?f1 ?f2)) (SET.FTN$source ?f1))
                  (= (first ((pairing-cone ?p) ?f1 ?f2)) ?f1)
                  (= (second ((pairing-cone ?p) ?f1 ?f2)) ?f2))))

(16) (KIF$function pairing)
     (KIF$signature pairing diagram CNG$function)
     (forall (?p (diagram ?p))
         (CNG$signature (pairing ?p)
          SET.FTN$function SET.FTN$function SET.FTN$function))
     (forall (?p (diagram ?p)
              ?f1 (SET.FTN$function ?f1) ?f2 (SET.FTN$function ?f2))
         (=> (and (= (SET.FTN$source ?f1) (SET.FTN$source ?f2))
                  (= (SET.FTN$target ?f1) (class1 ?p))
                  (= (SET.FTN$target ?f2) (class2 ?p)))
             (= ((pairing ?p) ?f1 ?f2)
                (mediator ((pairing-cone ?p) ?f1 ?f2)))))
```

- There is a CNG '`binary-product-opspan`' function that maps a pair (of classes) to an associated pullback opspan, whose opvertex is the `terminal` class and whose opfirst and opsecond functions are the unique functions for the pair of classes.

```
(17) (CNG$function binary-product-opspan)
```

```
(CNG$signature binary-product-opspan diagram SET.LIM.PBK$diagram)
(forall (?p (diagram ?p))
    (and (= (SET.LIM.PBK$class1 (binary-product-opspan ?p)) (class1 ?p))
         (= (SET.LIM.PBK$class2 (binary-product-opspan ?p)) (class2 ?p))
         (= (SET.LIM.PBK$opvertex (binary-product-opspan ?p)) terminal)
         (= (SET.LIM.PBK$opfirst (binary-product-opspan ?p))
            (unique (class1 ?p)))
         (= (SET.LIM.PBK$opsecond (binary-product-opspan ?p))
            (unique (class2 ?p)))))
```

- Using this opspan we can show that the notion of a product could be based upon pullbacks and the terminal object. We do this by proving the following theorem that the pullback of this opspan is the binary product class, and the pullback projections are the product projection functions.

```
(forall (?p (diagram ?p))
    (and (= (binary-product ?p)
            (SET.LIM.PBK$pullback (binary-product-opspan ?p)))
         (= (projection1 ?p)
            (SET.LIM.PBK$projection1 (binary-product-opspan2 ?p)))
         (= (projection2 ?p)
            (SET.LIM.PBK$projection2 (binary-product-opspan ?p)))))
```

- We can also prove the theorem that the product pairing of a pair (of classes) is the pullback pairing of the associated opspan.

```
(forall (?p (diagram ?p))
    (= (pairing ?p)
       (SET.LIM.PBK$pairing (binary-product-opspan ?p))))
```

- For any class C the unit laws for binary product say that the classes $1 \otimes C$ and C are isomorphic and that the graphs $C \otimes 1$ and C are isomorphic. The definitions for the appropriate bijection (isomorphisms), *left unit* $\lambda_C : 1 \otimes C \to C$ and *right unit* $\rho_C : C \otimes 1 \to C$, are as follows.

```
(18) (CNG$function right-diagram)
     (CNG$signature right-diagram SET$class diagram)

(19) (CNG$function right)
     (CNG$signature right SET$class SET.FTN$function)

     (forall (?c (SET$class ?c))
         (and (= (set1 (right-diagram ?c)) ?c)
              (= (set2 (right-diagram ?c)) SET.LIM$unit)
              (= (right ?c) (SET.LIM.PRD$projection1 (right-diagram ?c)))))

     (forall (?p ?x (pair ?p))
        (and (= (SET.FTN$composition (tau ?p) (tau (opposite ?p)))
                (SET.FTN$identity (binary-product (opposite ?p))))
             (= (SET.FTN$composition (tau (opposite ?p)) (tau ?p))
                (SET.FTN$identity (binary-product ?p)))))
```

- The product of the opposite of a pair is isomorphic to the product of the pair. This isomorphism is mediated by the *tau* or *twist* function.

```
(18) (CNG$function tau-cone)
     (CNG$signature tau-cone pair cone)
     (forall (?p (pair ?p))
        (and (= (vertex (tau-cone ?p)) (binary-product (opposite ?p)))
             (= (first (tau-cone ?p)) (projection2 (opposite ?p)))
             (= (second (tau-cone ?p)) (projection1 (opposite ?p)))))

(19) (CNG$function tau)
     (CNG$signature tau pair SET.FTN$function)
     (forall (?p (pair ?p))
        (and (= (SET.FTN$source (tau ?p)) (binary-product (opposite ?p)))
             (= (SET.FTN$target (tau ?p)) (binary-product ?p))))
     (forall (?p (pair ?p))
        (= (tau ?p) (mediator (tau-cone ?p))))
```

- The tau function is an isomorphism – the following theorem can be proven.

```
(forall (?p ?x (pair ?p))
   (and (= (SET.FTN$composition (tau ?p) (tau (opposite ?p)))
           (SET.FTN$identity (binary-product (opposite ?p))))
        (= (SET.FTN$composition (tau (opposite ?p)) (tau ?p))
           (SET.FTN$identity (binary-product ?p)))))
```

Function

SET.LIM.PRD.FTN

The product notion can be extended from pairs of classes to pairs of functions – in short, the product notion is quasi-functorial.

- The product operator extends from pairs of classes to pairs of functions (Figure 4). This is a specific *Cartesian Binary Product* function. Let '`pair`' be the SET namespace term that denotes the function pair collection.

```
(1) (CNG$conglomerate pair)

(2) (CNG$function source)
    (CNG$signature source pair SET.LIM.PRD$pair)

(3) (CNG$function target)
    (CNG$signature target pair SET.LIM.PRD$pair)

(4) (CNG$function function1)
    (CNG$signature function1 pair SET.FTN$function)
    (forall (?h (pair ?h))
        (and (= (SET.FTN$source (function1 ?h))
                (SET.LIM.PRD$class1 (source ?h))))
             (= (target (function1 ?h))
                (SET.LIM.PRD$class1 (target ?h)))))

(5) (CNG$function function2)
    (CNG$signature function2 pair function)
    (forall (?h (pair ?h))
       (and (= (source (function2 ?h))
               (SET.LIM.PRD$class2 (source ?h)))
            (= (target (function2 ?h))
               (SET.LIM.PRD$class2 (target ?h)))))

(6) (CNG$function binary-product)
    (CNG$signature binary-product pair SET.FTN$function)
    (forall (?h (pair ?h))
       (= (binary-product ?h)
          (the ?f (SET.FTN$function ?f))
              (and (= (SET.FTN$composition ?f (SET.LIM.PRD$projection1 (target ?h)))
                      (SET.FTN$composition
                          (SET.LIM.PRD$projection1 (source ?h)) ?f1))
                   (= (composition ?f (SET.LIM.PRD$projection2 (target ?h)))
                      (composition (SET.LIM.PRD$projection2 (source ?h)) ?f2))))))
```

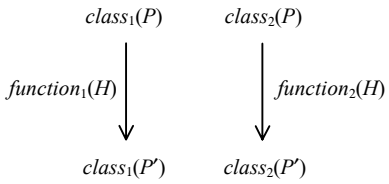


Figure 4: Function Pair

Equalizers

SET.LIM.EQU

An (binary) *equalizer* is a finite limit for a diagram of shape $\bullet \rightrightarrows \bullet$. Such a diagram (of classes and functions) is called a *parallel pair* of functions.

- A *parallel pair* is the appropriate base diagram for an equalizer. Each parallel pair consists of a pair of functions called *funtion1* and *function2* that share the same *source* and *target* classes. Let either '`diagram`' or '`parallel-pair`' be the SET namespace term that denotes the *Parallel Pair* collection. Parallel pairs are determined by their two component functions.

```
(1) (conglomerate diagram)
    (conglomerate parallel-pair)
    (= parallel-pair diagram)

(2) (CNG$function source)
    (CNG$signature source diagram SET$class)
```

```
(3) (CNG$function target)
    (CNG$signature target diagram SET$class)

(4) (CNG$function function1)
    (CNG$signature function1 diagram SET.FTN$function)

(5) (CNG$function function2)
    (CNG$signature function2 diagram SET.FTN$function)

    (forall (?p (diagram ?p))
        (and (= (SET.FTN$source (function1 ?p)) (source ?p))
             (= (SET.FTN$target (function1 ?p)) (target ?p))
             (= (SET.FTN$source (function2 ?p)) (source ?p))
             (= (SET.FTN$target (function2 ?p)) (target ?p))))

    (forall (?p (diagram ?p) ?q (diagram ?q))
        (=> (and (= (function1 ?p) (function1 ?q))
                 (= (function2 ?p) (function2 ?q)))
            (= ?p ?q)))
```

- *Equalizer Cones* are used to specify and axiomatize equalizers. Each equalizer cone (Figure 5) has an underlying *parallel-pair*, a *vertex* class, and a function called *function*, whose source class is the vertex and whose target class is the source class of the functions in the parallel-pair. The second function indicated in the diagram below is obviously not needed. An equalizer cone is the very special case of a cone over an parallel-pair. Let '`cone`' be the SET namespace term that denotes the *Equalizer Cone* collection.

```
(6) (CNG$conglomerate cone)

(7) (CNG$function cone-diagram)
    (CNG$signature cone-diagram cone diagram)

(8) (CNG$function vertex)
    (CNG$signature vertex cone SET$class)

(9) (CNG$function function)
    (CNG$signature function cone SET.FTN$function)
    (forall (?r (cone ?r))
       (and (= (SET.FTN$source (function ?r)) (vertex ?r))
            (= (SET.FTN$target (function ?r))
               (source (cone-diagram ?r)))))

    (forall (?r (cone ?r))
       (= (SET.FTN$composition (function ?r) (function1 (cone-diagram ?r)))
          (SET.FTN$composition (function ?r) (function2 (cone-diagram ?r)))))
```

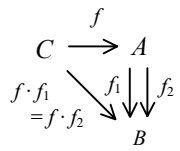


Figure 5: Equalizer Cone

- There is a unary CNG function '`limiting-cone`' that maps a parallel-pair to its equalizer (limiting equalizer cone) (Figure 6). Axiom (*) asserts that this function is total. This, along with the universality of the mediator function, implies that an equalizer exists for any parallel-pair. The vertex of the equalizer cone is a specific *Cartesian Equalizer* class given by the CNG function '`equalizer`'. It comes equipped with a CNG canonical equalizing function '`canon`'. This notation is for convenience of reference. It is used for equalizers in general. Axiom (#) ensures that this equalizer is specific – that it is exactly the subclass of the source class on which the two functions agree.

```
(10) (CNG$function limiting-cone)
     (CNG$signature limiting-cone diagram cone)
 (*) (forall (?p (diagram ?p))
         (exists (?r (cone ?r))
             (= (limiting-cone ?p) ?r)))
     (forall (?p (diagram ?p))
        (= (cone-diagram (limiting-cone ?p)) ?p))

(11) (CNG$function limit)
     (CNG$function equalizer)
     (= limit equalizer)
     (CNG$signature limit diagram SET$class)
     (forall (?p (diagram ?p))
```

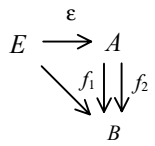


Figure 6: Limiting Cone

```
          (= (limit ?p) (vertex (limiting-cone ?p))))

(12) (CNG$function canon)
     (CNG$signature canon diagram SET.FTN$function)
     (forall (?p (diagram ?p))
        (= (canon ?p) (function (limiting-cone ?p))))

 (#) (forall (?p (diagram ?p))
         (and (SET$subclass (limit ?p) (source ?p))
              (forall (?x ((limit ?p) ?x))
                  (= ((canon ?p) ?x) ?x)))
```

- There is a *mediator* function from the vertex of a cone over a parallel pair (of functions) to the equalizer of the parallel pair. This is the unique function that commutes with equalizing canon and cone function. We use a KIF definite description abbreviation to define this. Existence and uniqueness represents the universality of the equalizer operator.

```
(13) (CNG$function mediator)
     (CNG$signature mediator cone SET.FTN$function)
     (forall (?r (cone ?r))
        (= (mediator ?r)
           (the (?f (SET.FTN$function ?f))
                (and (= (SET.FTN$source ?f) (vertex ?r))
                     (= (SET.FTN$target ?f) (limit (cone-diagram ?r)))
                     (= (SET.FTN$composition ?f (canon (cone-diagram ?r)))
                        (function ?r)))))))
```

- For any function $f : A \to B$ there is a *kernel* equivalence relation on the source set A.

```
(14) (CNG$function kernel-diagram)
     (CNG$signature kernel-diagram SET.FTN$function parallel-pair)
     (forall (?f (SET.FTN$function ?f))
         (and (source (kernel-diagram ?f)) (SET.FTN$source ?f))
              (target (kernel-diagram ?f)) (SET.FTN$target ?f))
              (function1 (kernel-diagram ?f)) ?f)
              (function2 (kernel-diagram ?f)) ?f)))

(15) (CNG$function kernel)
     (CNG$signature kernel SET.FTN$function REL.ENDO$equivalence-relation)
     (forall (?f (SET.FTN$function ?f))
        (and (= (REL.ENDO$object (kernel ?f))
                (source ?f))
             (= (REL.ENDO$extent (kernel ?f))
                (equalizer (kernel-diagram ?f)))))
```

Subequalizers

`SET.LIM.SEQU`

A *subequalizer* is a lax equalizer – a lax limit for a lax diagram consisting of a parallel pair of functions whose target is an order.

- A *lax parallel pair* $f_1, f_2 : A \to \boldsymbol{B} = \langle B, \leq \rangle$ is the appropriate base diagram for a subequalizer. A lax parallel pair consists of a parallel pair of functions whose target class is the base class of an order. Let either '`lax-diagram`' or '`lax-parallel-pair`' be the SET namespace term that denotes the *Lax Parallel Pair* collection.

```
(1) (conglomerate lax-diagram)
    (conglomerate lax-parallel-pair)
    (= lax-parallel-pair lax-diagram)

(2) (CNG$function order)
    (CNG$signature order lax-diagram ORD$order)

(3) (CNG$function source)
    (CNG$signature source lax-diagram SET$class)

(4) (CNG$function function1)
    (CNG$signature function1 lax-diagram SET.FTN$function)

(5) (CNG$function function2)
```

```
(CNG$signature function2 lax-diagram SET.FTN$function)

(forall (?p (lax-diagram ?p))
    (and (= (SET.FTN$source (function1 ?p)) (source ?p))
         (= (SET.FTN$source (function2 ?p)) (source ?p))
         (= (SET.FTN$target (function1 ?p)) (ORD$class (order ?p)))
         (= (SET.FTN$target (function2 ?p)) (ORD$class (order ?p)))))
```

- Any equalizer diagram (parallel pair) embeds as a subequalizer diagram (lax parallel pair), where the order has the identity order relation.

```
(6) (CNG$function lax)
    (CNG$signature lax SET.LIM.EQU$diagram lax-diagram)

    (forall (?p (SET.LIM.EQU$diagram ?p))
        (and (= (source (lax ?p)) (SET.LIM.EQU$source ?p))
             (= (order (lax ?p)) (ORD$identity (SET.LIM.EQU$target ?p)))
             (= (function1 (lax ?p)) (SET.LIM.EQU$function1 ?p))
             (= (function2 (lax ?p)) (SET.LIM.EQU$function2 ?p))))
```

- The underlying *parallel pair* of any lax parallel pair (subequalizer diagram) is named. The underlying parallel pair of the lax embedding of a strict parallel pair is itself. Lax parallel pairs are determined by their target order and parallel pair.

```
(7) (CNG$function parallel-pair)
    (CNG$signature parallel-pair lax-diagram SET.LIM.EQU$diagram)

    (forall (?p (lax-diagram ?p))
        (and (= (SET.LIM.EQU$source (parallel-pair ?p)) (source ?p))
             (= (SET.LIM.EQU$target (parallel-pair ?p)) (ORD$class (order ?p)))
             (= (SET.LIM.EQU$function1 (parallel-pair ?p)) (function1 ?p))
             (= (SET.LIM.EQU$function2 (parallel-pair ?p)) (function2 ?p))))

    (forall (?p (SET.LIM.EQU$diagram ?p))
        (= (parallel-pair (lax ?p)) ?p))

    (forall (?p (lax-diagram ?p) ?q (lax-diagram ?q))
        (=> (and (= (order ?p) (order ?q))
                 (= (parallel-pair ?p) (parallel-pair ?q)))
            (= ?p ?q)))
```

- *Subequalizer Cones* are used to specify and axiomatize equalizers. Each subequalizer cone (Figure 7) has an *order*, and underlying *parallel-pair* whose target is that order, a *vertex* class, and a function called *function*, whose source class is the vertex and whose target class is the source class of the functions in the parallel-pair. A subequalizer cone is the very special case of a lax cone over an lax-parallel-pair. The function composition is only required to be an inequality, not an equality. Let '`lax-cone`' be the SET namespace term that denotes the *Subequalizer Cone* collection.

```
(8) (CNG$conglomerate lax-cone)

(9) (CNG$function lax-cone-diagram)
    (CNG$signature lax-cone-diagram lax-cone lax-diagram)

(10) (CNG$function vertex)
    (CNG$signature vertex lax-cone SET$class)

(11) (CNG$function function)
     (CNG$signature function lax-cone SET.FTN$function)

     (forall (?r (lax-cone ?r))
        (and (= (SET.FTN$source (function ?r))
                (vertex ?r))
             (= (SET.FTN$target (function ?r))
                (source (lax-cone-diagram ?r)))))

     (forall (?r (lax-cone ?r))
         ((ORD$relation (order (lax-cone-diagram ?r)))
              ((function1 (lax-cone-diagram ?r)) ((function ?r) ?x))
              ((function2 (lax-cone-diagram ?r)) ((function ?r) ?x))))
```

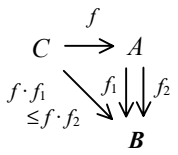


Figure 7: Subequalizer Cone

- There is a unary CNG function '`limiting-lax-cone`' that maps a lax-parallel-pair $f_1, f_2 : A \to \boldsymbol{B} = \langle B, \leq \rangle$ to its subequalizer (lax limiting subequalizer cone) (Figure 8). Axiom (*) asserts that this function is total. This, along with the universality of the mediator function, implies that an subequalizer exists for any lax-parallel-pair. The vertex of the subequalizer cone is a specific *Cartesian Subequalizer* class $\{a \in A \mid f_1(a) \leq f_2(a)\} \subseteq A$ given by the CNG function '`subequalizer`'. It comes equipped with a CNG canonical subequalizing function '`subcanon`', which is the inclusion of the subequalizer class into source class A. This notation is for convenience of reference. It is used for subequalizers in general. Axiom (#) ensures that this subequalizer is specific – that it is exactly the subclass of the source class on which the two functions are ordered. Obviously, equalizers are a special case of subequalizers – just use the lax embedding of the equalizer diagram.

```
(12) (CNG$function limiting-lax-cone)
     (CNG$signature limiting-lax-cone lax-diagram lax-cone)
 (*) (forall (?p (lax-diagram ?p))
         (exists (?r (lax-cone ?r))
             (= (limiting-lax-cone ?p) ?r)))
     (forall (?p (lax-diagram ?p))
        (= (lax-cone-diagram (limiting-lax-cone ?p)) ?p))
```

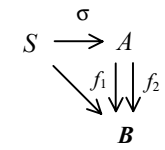


Figure 8: Lax Limiting Cone

```
(13) (CNG$function lax-limit)
     (CNG$function subequalizer)
     (= subequalizer lax-limit)
     (CNG$signature subequalizer lax-diagram SET$class)
     (forall (?p (lax-diagram ?p))
        (= (subequalizer ?p)
           (vertex (limiting-lax-cone ?p))))

(14) (CNG$function subcanon)
     (CNG$signature subcanon lax-diagram SET.FTN$function)
     (forall (?p (lax-diagram ?p))
        (= (subcanon ?p) (function (limiting-lax-cone ?p))))

 (#) (forall (?p (lax-diagram ?p))
         (and (SET$subclass (subequalizer ?p) (source ?p))
              (forall (?x ((subequalizer ?p) ?x))
                  (= ((subcanon ?p) ?x) ?x)))
```

- There is a *mediator* function from the vertex of a lax cone over a lax-parallel-pair to the subequalizer of the lax-parallel-pair. This is the unique function that laxly commutes with subequalizing subcanon and lax-cone function. We use a KIF definite description abbreviation to define this. Existence and uniqueness represents the universality of the subequalizer operator.

```
(15) (CNG$function mediator)
     (CNG$signature mediator lax-cone SET.FTN$function)
     (forall (?r (lax-cone ?r))
        (= (mediator ?r)
           (the (?f (SET.FTN$function ?f))
                (and (= (SET.FTN$source ?f) (vertex ?r))
                     (= (SET.FTN$target ?f) (subequalizer (lax-cone-diagram ?r)))
                     (= (SET.FTN$composition ?f (subcanon (lax-cone-diagram ?r)))
                        (function ?r)))))))
```

- There is one special kind of subequalizer that deserves mention. For any order $\boldsymbol{A} = \langle B, \leq \rangle$ the *suborder* of $\boldsymbol{A}$ is the subequalizer for the pair of identity functions $id_A, id_A : A \to \boldsymbol{A} = \langle A, \leq \rangle$.

```
(16) (CNG$function suborder-lax-diagram)
     (CNG$signature suborder-lax-diagram ORD$order lax-diagram)
     (forall (?o (ORD$order ?o))
         (and (= (order (suborder-lax-diagram ?o)) ?o)
              (= (source (suborder-lax-diagram ?o)) (ORD$class ?o))
              (= (function1 (suborder-lax-diagram ?o))
                 (SET.FTN$identity (ORD$class ?o)))
              (= (function2 (suborder-lax-diagram ?o))
                 (SET.FTN$identity (ORD$class ?o))))))

(17) (CNG$function suborder)
     (CNG$signature suborder ORD$order SET$class)
```

```
(forall (?o (ORD$order ?o))
    (= (suborder ?o) (subequalizer (suborder-lax-diagram ?o))))
```

Pullbacks

SET.LIM.PBK

A *pullback* is a finite limit for a diagram of shape $\bullet \rightarrow \bullet \leftarrow \bullet$. Such a diagram (of classes and functions) is called an *opspan*.

- An *opspan* is the appropriate base diagram for a pullback. An opspan is the opposite of an span. Each opspan consists of a pair of functions called *opfirst* and *opsecond*. These are required to have a common target class, denoted as the *opvertex*. Let either '`diagram`' or '`opspan`' be the SET namespace term that denotes the *Opspan* collection. Opspans are determined by their pair of component functions.

```
(1) (CNG$conglomerate diagram)
    (CNG$conglomerate opspan)
    (= opspan diagram)

(2) (CNG$function class1)
    (CNG$signature class1 diagram SET$class)

(3) (CNG$function class2)
    (CNG$signature class2 diagram SET$class)

(4) (CNG$function opvertex)
    (CNG$signature opvertex diagram SET$class)

(5) (CNG$function opfirst)
    (CNG$signature opfirst diagram SET.FTN$function)

(6) (CNG$function opsecond)
    (CNG$signature opsecond diagram SET.FTN$function)

    (forall (?s (diagram ?s))
       (and (= (SET.FTN$source (opfirst ?s)) (class1 ?s))
            (= (SET.FTN$source (opsecond ?s)) (class2 ?s))
            (= (SET.FTN$target (opfirst ?s)) (opvertex ?s))
            (= (SET.FTN$target (opsecond ?s)) (opvertex ?s))))

    (forall (?s (diagram ?s) ?t (diagram ?t))
       (=> (and (= (opfirst ?s) (opfirst ?t))
                (= (opsecond ?s) (opsecond ?t)))
           (= ?s ?t)))
```

- The *pair* of source classes (prefixing discrete diagram) of any opspan (pullback diagram) is named.

```
(7) (CNG$function pair)
    (CNG$signature pair diagram SET.LIM.PRD$diagram)
    (forall (?s (diagram ?s))
        (and (SET.LIM.PRD$class1 (pair ?s)) (class1 ?s))
             (SET.LIM.PRD$class2 (pair ?s)) (class2 ?s))))
```

- Every opspan has an opposite.

```
(8) (CNG$function opposite)
    (CNG$signature opposite opspan opspan)

    (forall (?s (opspan ?s))
       (and (= (class1 (opposite ?s)) (class2 ?s))
            (= (class2 (opposite ?s)) (class1 ?s))
            (= (opvertex (opposite ?s)) (opvertex ?s))
            (= (opfirst (opposite ?s)) (opsecond ?s))
            (= (opsecond (opposite ?s)) (opfirst ?s))))
```

- The opposite of the opposite is the original opspan – the following theorem can be proven.

```
    (forall (?s (opspan ?s))
       (= (opposite (opposite ?s)) ?s))
```

- *Pullback cones* are used to specify and axiomatize pullbacks. Each pullback cone (Figure 9) has an underlying *opspan*, a *vertex* class, and a pair of functions called *first* and *second*, whose common source class is the vertex and whose target classes are the source classes of the functions in the opspan. The first and second functions form a commutative diagram with the opspan. A pullback cone is the very special case of a cone over an opspan. Let 'cone' be the SET namespace term that denotes the *Pullback Cone* collection.

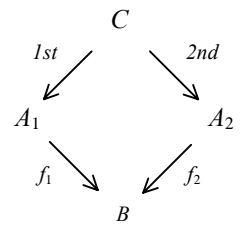


Figure 9: Pullback Cone

```
(9) (CNG$conglomerate cone)

(10) (CNG$function cone-diagram)
     (CNG$signature cone-diagram cone diagram)

(11) (CNG$function vertex)
     (CNG$signature vertex cone SET$class)

(12) (CNG$function first)
     (CNG$signature first cone SET.FTN$function)
     (forall (?r (cone ?r))
        (and (= (SET.FTN$source (first ?r)) (vertex ?r))
             (= (SET.FTN$target (first ?r)) (class1 (cone-diagram ?r)))))

(13) (CNG$function second)
     (CNG$signature second cone SET.FTN$function)
     (forall (?r (cone ?r))
        (and (= (SET.FTN$source (second ?r)) (vertex ?r))
             (= (SET.FTN$target (second ?r)) (class2 (cone-diagram ?r)))))

     (forall (?r (cone ?r))
        (= (SET.FTN$composition (first ?r) (opfirst (cone-diagram ?r)))
           (SET.FTN$composition (second ?r) (opsecond (cone-diagram ?r)))))
```

- There is a unary CNG function 'limiting-cone' that maps an opspan to its pullback (limiting pullback cone) (Figure 10). Axiom (*) asserts that this function is total. This, along with the universality of the mediator function, implies that a pullback exists for any opspan. The vertex of the pullback cone is a specific *Cartesian Pullback* class given by the CNG function 'pullback'. It comes equipped with two CNG projection functions 'projection1' and 'projection2'. This notation is for convenience of reference. It is used for pullbacks in general. Axiom (#) ensures that this pullback is specific – that it is exactly the subclass of the Cartesian product on which the opfirst and opsecond functions agree. Finally, there is a unary CNG function 'relation' that alternatively represents the pullback as a large relation.

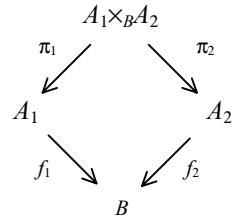


Figure 10: Limiting Cone

```
(14) (CNG$function limiting-cone)
     (CNG$signature limiting-cone diagram cone)
 (*) (forall (?s (diagram ?s))
         (exists (?r (cone ?r))
             (= (limiting-cone ?s) ?r)))
     (forall (?s (diagram ?s))
        (= (cone-diagram (limiting-cone ?s)) ?s))

(15) (CNG$function limit)
     (CNG$function pullback)
     (= pullback limit)
     (CNG$signature limit diagram SET$class)
     (forall (?s (diagram ?s))
        (= (limit ?s) (vertex (limiting-cone ?s))))

(16) (CNG$function projection1)
     (CNG$signature projection1 diagram SET.FTN$function)
     (forall (?s (diagram ?s))
```

```
          (and (= (SET.FTN$source (projection1 ?s)) (limit ?s))
               (= (SET.FTN$target (projection1 ?s)) (class1 ?s))
               (= (projection1 ?s) (first (limiting-cone ?s)))))

(17) (CNG$function projection2)
     (CNG$signature projection2 diagram SET.FTN$function)
     (forall (?s (diagram ?s))
        (and (= (SET.FTN$source (projection2 ?s)) (limit ?s))
             (= (SET.FTN$target (projection2 ?s)) (class2 ?s))
             (= (projection2 ?s) (second (limiting-cone ?s)))))

 (#) (forall (?s (diagram ?s))
         (and (SET$subclass (limit ?s) (SET.LIM.PRD$binary-product (pair ?s)))
              (forall (?x1 ?x2 ((limit ?s) [?x1 ?x2]))
                  (and (= ((projection1 ?s) [?x1 ?x2]) ?x1)
                       (= ((projection2 ?s) [?x1 ?x2]) ?x2)))))

(18) (CNG$function relation)
     (CNG$signature relation diagram REL$relation)
     (forall (?s (diagram ?s))
        (and (= (REL$object1 (relation ?s)) (class1 ?s))
             (= (REL$object2 (relation ?s)) (class2 ?s))
             (= (REL$extent (relation ?s)) (limit ?s))))
```

o There is a *mediator* function from the vertex of a cone over an opspan to the pullback of the opspan. This is the unique function that commutes with first and second. We use a KIF definite description abbreviation to define this. Existence and uniqueness represents the universality of the pullback operator. We have also introduced a convenience term '`pairing`'. With an opspan parameter, the binary CNG function '`(pairing ?s)`' maps a pair of SET functions, that form a cone over the opspan, to their mediator (pairing) function.

```
(19) (CNG$function mediator)
     (CNG$signature mediator cone SET.FTN$function)
     (forall (?r (cone ?r))
        (= (mediator ?r)
           (the (?f (SET.FTN$function ?f))
                (and (= (SET.FTN$source ?f) (vertex ?r))
                     (= (SET.FTN$target ?f) (limit (cone-diagram ?r)))
                     (= (SET.FTN$composition ?f (projection1 (cone-diagram ?r)))
                        (first ?r))
                     (= (SET.FTN$composition ?f (projection2 (cone-diagram ?r)))
                        (second ?r)))))))

(20) (KIF$function pairing-cone)
     (KIF$signature pairing-cone opspan CNG$function)
     (forall (?s (opspan ?s))
        (and (CNG$signature (pairing-cone ?s)
              SET.FTN$function SET.FTN$function cone)
             (=> (exists (?f1 ?f2 (SET.FTN$function ?f1) (SET.FTN$function ?f2)
                          ?r (cone ?r))
                     (= (((pairing-cone ?s) [?f1 ?f2]) ?r))
                 (and (= (SET.FTN$source ?f1) (SET.FTN$source ?f2))
                      (= (SET.FTN$composition ?f1 (opfirst ?s))
                         (SET.FTN$composition ?f2 (opsecond ?s)))))))
     (forall (?s (opspan ?s)
              ?f1 (SET.FTN$function ?f1) ?f2 (SET.FTN$function ?f2))
        (=> (and (= (SET.FTN$source ?f1) (SET.FTN$source ?f2))
                 (= (SET.FTN$composition ?f1 (opfirst ?s))
                    (SET.FTN$composition ?f2 (opsecond ?s)))
            (and (= (cone-opspan ((pairing-cone ?s) ?f1 ?f2)) ?s)
                 (= (vertex ((pairing-cone ?s) ?f1 ?f2)) (SET.FTN$source ?f1))
                 (= (first ((pairing-cone ?s) ?f1 ?f2)) ?f1)
                 (= (second ((pairing-cone ?s) ?f1 ?f2)) ?f2))))

(21) (KIF$function pairing)
     (KIF$signature pairing opspan CNG$function)
     (forall (?s (opspan ?s))
        (CNG$signature (pairing ?s)
         SET.FTN$function SET.FTN$function SET.FTN$function))
```

```
(forall (?s (opspan ?s)
        ?f1 (SET.FTN$function ?f1) ?f2 (SET.FTN$function ?f2))
   (=> (and (= (SET.FTN$source ?f1) (SET.FTN$source ?f2))
            (= (SET.FTN$composition ?f1 (opfirst ?s))
               (SET.FTN$composition ?f2 (opsecond ?s)))
       (= ((pairing ?s) ?f1 ?f2)
          (mediator ((pairing-cone ?s) ?f1 ?f2)))))
```

- Associated with any class opspan $S = (f_1 : A_1 \to B,\ f_2 : A_2 \to B)$ with pullback $1^{st} : A_1 \times_B A_2 \to A_1$, $2^{nd} : A_1 \times_B A_2 \to A_2$ are five fiber functions (Diagram 2), the last two of which are derived,

$$\phi^S : B \to \wp(A_1 \times_B A_2)$$
$$\phi^S{}_1 : B \to \wp A_1 \qquad \phi^S{}_{12} : A_1 \to \wp A_2$$
$$\phi^S{}_2 : B \to \wp A_2 \qquad \phi^S{}_{21} : A_2 \to \wp A_1$$

five embedding functionals, the last two of which are derived,

$$\iota^S{}_b : \phi^S(b) \to A_1 \times_B A_2$$
$$\iota^S{}_{1b} : \phi^S{}_1(b) \to A_1 \qquad \iota^S{}_{12a1} : \phi^S{}_{12}(a_1) = \phi_2{}^S(f_1(a_1)) \to \phi^S(f_1(a_1))$$
$$\iota^S{}_{2b} : \phi^S{}_2(b) \to A_2 \qquad \iota^S{}_{21a2} : \phi^S{}_{21}(a_2) = \phi_1{}^S(f_2(a_2)) \to \phi^S(f_2(a_2))$$

and two projection functionals

$$\pi^S{}_{1b} : \phi^S(b) \to \phi^S{}_1(b)$$
$$\pi^S{}_{2b} : \phi^S(b) \to \phi^S{}_2(b)$$

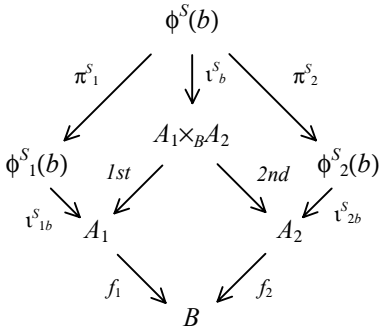


Diagram 2: Pullback Fibers

Here are the pointwise definitions.

$$\phi^S(b) = \{(a_1, a_2) \in A_1 \times_B A_2 \mid f_1(a_1) = b = f_2(a_2)\} \subseteq A_1 \times_B A_2$$
$$\phi^S{}_1(b) = \{a_1 \in A_1 \mid f_1(a_1) = b\} \subseteq A_1 \qquad \phi^S{}_{12}(a_1) = \{a_2 \in A_2 \mid f_1(a_1) = f_2(a_2)\} = \phi^S{}_2(f_1(a_1))$$
$$\phi^S{}_2(b) = \{a_2 \in A_2 \mid b = f_2(a_2)\} \subseteq A_2 \qquad \phi^S{}_{21}(a_2) = \{a_1 \in A_1 \mid f_1(a_1) = f_2(a_2)\} = \phi^S{}_1(f_2(a_2))$$

Using the fiber (point-wise power) functional $(\text{-})^{-1}$, we can define these as follows.

$$\phi^S = (1^{st} \cdot f_1)^{-1}$$
$$\phi^S{}_1 = f_1^{-1} \qquad \phi^S{}_{12} = f_1 \cdot f_2^{-1}$$
$$\phi^S{}_2 = f_2^{-1} \qquad \phi^S{}_{21} = f_2 \cdot f_1^{-1}$$

We clearly have the identifications: $f_1 \cdot \phi^S{}_2 = \phi^S{}_{12}$ and $f_2 \cdot \phi^S{}_1 = \phi^S{}_{21}$.

```
(22) (CNG$function fiber)
     (CNG$signature fiber opspan SET.FTN$function)
     (forall (?s (opspan ?s))
        (and (= (SET.FTN$source (fiber ?s))
                (opvertex ?s))
             (= (SET.FTN$target (fiber ?s))
                (SET$power (pullback ?s)))
             (= (fiber ?s)
                (SET.FTN$fiber
                    (SET.FTN$composition (projection1 ?s) (opfirst ?s))))))

(23) (CNG$function fiber1)
     (CNG$signature fiber1 opspan SET.FTN$function)
     (forall (?s (opspan ?s))
        (and (= (SET.FTN$source (fiber1 ?s)) (opvertex ?s))
             (= (SET.FTN$target (fiber1 ?s)) (SET$power (class1 ?s)))
             (= (fiber1 ?s) (SET.FTN$fiber (opfirst ?s)))))

(24) (CNG$function fiber2)
     (CNG$signature fiber2 opspan SET.FTN$function)
     (forall (?s (opspan ?s))
        (and (= (SET.FTN$source (fiber2 ?s)) (opvertex ?s))
```

```
            (= (SET.FTN$target (fiber2 ?s)) (SET$power (class2 ?s)))
            (= (fiber2 ?s) (SET.FTN$fiber (opsecond ?s)))))

(25) (CNG$function fiber12)
     (CNG$signature fiber12 opspan SET.FTN$function)
     (forall (?s (opspan ?s))
        (and (= (SET.FTN$source (fiber12 ?s)) (class1 ?s))
             (= (SET.FTN$target (fiber12 ?s)) (SET$power (class2 ?s)))
             (= (fiber12 ?s) (SET.FTN$composition (opfirst ?s) fiber2))))

(26) (CNG$function fiber21)
     (CNG$signature fiber21 opspan SET.FTN$function)
     (forall (?s (opspan ?s))
        (and (= (SET.FTN$source (fiber21 ?s)) (class2 ?s))
             (= (SET.FTN$target (fiber21 ?s)) (SET$power (class1 ?s)))
             (= (fiber21 ?s) (SET.FTN$composition (opsecond ?s) fiber1))))

(27) (KIF$function fiber-embedding)
     (KIF$signature fiber-embedding opspan CNG$function)
     (forall (?s (opspan ?s))
        (CNG$signature (fiber-embedding ?s) (opvertex ?s) SET.FTN$function))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y))
        (and (= (SET.FTN$source ((fiber-embedding ?s) ?y))
                ((fiber ?s) ?y))
             (= (SET.FTN$target ((fiber-embedding ?s) ?y))
                (pullback ?s))))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y)
             ?z (((fiber ?s) ?y) ?z))
        (= (((fiber-embedding ?s) ?y) ?z) ?z))

(28) (KIF$function fiber1-embedding)
     (KIF$signature fiber1-embedding opspan CNG$function)
     (forall (?s (opspan ?s))
        (CNG$signature (fiber1-embedding ?s) (opvertex ?s) SET.FTN$function))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y))
        (and (= (SET.FTN$source ((fiber1-embedding ?s) ?y)) ((fiber1 ?s) ?y))
             (= (SET.FTN$target ((fiber1-embedding ?s) ?y)) (class1 ?s))))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y)
             ?x1 (((fiber1 ?s) ?y) ?x1))
        (= (((fiber1-embedding ?s) ?y) ?x1) ?x1))

(29) (KIF$function fiber2-embedding)
     (KIF$signature fiber2-embedding opspan CNG$function)
     (forall (?s (opspan ?s))
        (CNG$signature (fiber2-embedding ?s) (opvertex ?s) SET.FTN$function))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y))
        (and (= (SET.FTN$source ((fiber2-embedding ?s) ?y)) ((fiber2 ?s) ?y))
             (= (SET.FTN$target ((fiber2-embedding ?s) ?y)) (class2 ?s))))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y)
             ?x2 (((fiber2 ?s) ?y) ?x2))
        (= (((fiber2-embedding ?s) ?y) ?x2) ?x2))

(30) (KIF$function fiber12-embedding)
     (KIF$signature fiber12-embedding opspan CNG$function)
     (forall (?s (opspan ?s))
        (CNG$signature (fiber12-embedding ?s) (class1 ?s) SET.FTN$function))
     (forall (?s (opspan ?s)
              ?x1 ((class1 ?s) ?x1))
        (and (= (SET.FTN$source ((fiber12-embedding ?s) ?x1))
                ((fiber12 ?s) ?x1))
             (= (SET.FTN$target ((fiber12-embedding ?s) ?x1))
                ((fiber ?s) ((opfirst ?s) ?x1)) )))
     (forall (?s (opspan ?s)
             ?x1 ((class1 ?s) ?x1)
```

```
            ?x2 (((fiber12 ?s) ?x1) ?x2))
         (= (((fiber12-embedding ?s) ?x1) ?x2) [?x1 ?x2]))

(31) (KIF$function fiber21-embedding)
     (KIF$signature fiber21-embedding opspan CNG$function)
     (forall (?s (opspan ?s))
        (CNG$signature (fiber21-embedding ?s) (class2 ?s) SET.FTN$function))
     (forall (?s (opspan ?s)
             ?x2 ((class2 ?s) ?x2))
        (and (= (SET.FTN$source ((fiber21-embedding ?s) ?x2))
                ((fiber21 ?s) ?x2))
             (= (SET.FTN$target ((fiber21-embedding ?s) ?x2))
                ((fiber ?s) ((opsecond ?s) ?x2)))))
     (forall (?s (opspan ?s)
             ?x2 ((class2 ?s) ?x2)
             ?x1 (((fiber21 ?s) ?x2) ?x1))
        (= (((fiber21-embedding ?s) ?x2) ?x1) [?x1 ?x2]))

(32) (KIF$function fiber1-projection)
     (KIF$signature fiber1-projection opspan CNG$function)
     (forall (?s (opspan ?s))
        (CNG$signature (fiber1-projection ?s) (opvertex ?s) SET.FTN$function))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y))
        (and (= (SET.FTN$source ((fiber1-projection ?s) ?y))
                ((fiber ?s) ?y))
             (= (SET.FTN$target ((fiber1-projection ?s) ?y))
                ((fiber1 ?s) ?y))))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y)
             ?x1 ?x2 (((fiber ?s) ?y) [?x1 ?x2]))
        (= (((fiber1-projection ?s) ?y) [?x1 ?x2]) ?x1))

(33) (KIF$function fiber2-projection)
     (KIF$signature fiber2-projection opspan CNG$function)
     (forall (?s (opspan ?s))
        (CNG$signature (fiber2-projection ?s) (opvertex ?s) SET.FTN$function))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y))
        (and (= (SET.FTN$source ((fiber2-projection ?s) ?y))
                ((fiber ?s) ?y))
             (= (SET.FTN$target ((fiber2-projection ?s) ?y))
                ((fiber2 ?s) ?y))))
     (forall (?s (opspan ?s)
             ?y ((opvertex ?s) ?y)
             ?x1 ?x2 (((fiber ?s) ?y) [?x1 ?x2]))
        (= (((fiber2-projection ?s) ?y) [?x1 ?x2]) ?x2))
```

- For any function $f : A \to B$ there is a *kernel-pair* equivalence relation on the source set A.

```
(34) (CNG$function kernel-pair-diagram)
     (CNG$signature kernel-pair-diagram SET.FTN$function opspan)
     (forall (?f (SET.FTN$function ?f))
         (and (class1 (kernel-pair-diagram ?f)) (SET.FTN$source ?f))
              (class2 (kernel-pair-diagram ?f)) (SET.FTN$source ?f))
              (opvertex (kernel-pair-diagram ?f)) (SET.FTN$target ?f))
              (opfirst (kernel-pair-diagram ?f)) ?f)
              (opsecond (kernel-pair-diagram ?f)) ?f)))

(35) (CNG$function kernel-pair)
     (CNG$signature kernel-pair SET.FTN$function REL.ENDO$equivalence-relation)
     (forall (?f (SET.FTN$function ?f))
        (and (= (REL.ENDO$class (kernel-pair ?f))
                (source ?f))
             (= (REL.ENDO$extent (kernel-pair ?f))
                (pullback (kernel-pair-diagram ?f)))))
```

- The pullback of the opposite of an opspan is isomorphic to the pullback of the opspan. This isomorphism is mediated by the *tau* or *twist* function.

```
(36) (CNG$function tau-cone)
```

```
        (CNG$signature tau-cone opspan cone)
        (forall (?s (opspan ?s))
           (and (= (opspan (tau-cone ?s)) ?s)
                (= (vertex (tau-cone ?s)) (pullback (opposite ?s)))
                (= (first (tau-cone ?s)) (projection2 (opposite ?s)))
                (= (second (tau-cone ?s)) (projection1 (opposite ?s)))))

    (37) (CNG$function tau)
         (CNG$signature tau opspan SET.FTN$function)
         (forall (?s (opspan ?s))
            (and (= (SET.FTN$source (tau ?s)) (pullback (opposite ?s)))
                 (= (SET.FTN$target (tau ?s)) (pullback ?s))))
         (forall (?s (opspan ?s))
            (= (tau ?s) (mediator (tau-cone ?s))))
```

- The tau function is an isomorphism – the following theorem can be proven.

```
        (forall (?s ?x (opspan ?s))
           (and (= (SET.FTN$composition (tau ?s) (tau (opposite ?s)))
                   (SET.FTN$identity (pullback (opposite ?s))))
                (= (SET.FTN$composition (tau (opposite ?s)) (tau ?s))
                   (SET.FTN$identity (pullback ?s)))))
```

Opspan Morphisms

`SET.LIM.PBK.MOR`

- An *opspan morphism* $H : S \to S'$ from a source opspan S to a target opspan S' consists of a triple of functions called *class1*, *class2* and *opvertex*. These (Figure 11) have source and target opspans, and are required to commute with the opfirst and opsecond functions of source and target. Let '`opspan-morphism`' be the SET namespace term that denotes the *Opspan Morphism* collection.

```
(1) (CNG$conglomerate opspan-morphism)

(2) (CNG$function source)
    (CNG$signature source opspan-morphism opspan)

(3) (CNG$function target)
    (CNG$signature target opspan-morphism opspan)

(4) (CNG$function opvertex)
    (CNG$signature opvertex opspan-morphism function)

(5) (CNG$function class1)
    (CNG$signature class1 opspan-morphism SET.FTN$function)

(6) (CNG$function class2)
    (CNG$signature class2 opspan-morphism function)
```

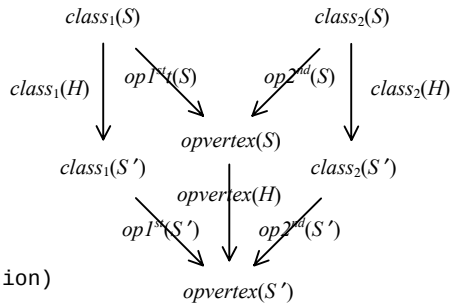


Figure 11: Opspan Morphism

```
    (forall (?h (opspan-morphism ?h))
       (and (= (SET.FTN$source (opvertex ?h)) (opvertex (source ?h)))
            (= (SET.FTN$target (opvertex ?h)) (opvertex (target ?h)))
            (= (SET.FTN$source (class1 ?h)) (class1 (source ?h)))
            (= (SET.FTN$target (class1 ?h)) (class1 (target ?h)))
            (= (SET.FTN$source (class2 ?h)) (class2 (source ?h)))
            (= (SET.FTN$target (class2 ?h)) (class2 (target ?h)))
            (= (SET.FTN$composition (opfirst (source ?h)) (opvertex ?h))
               (SET.FTN$composition (class1 ?h) (opfirst (target ?h))))
            (= (SET.FTN$composition (opsecond (source ?h)) (opvertex ?h))
               (SET.FTN$composition (class2 ?h) (opsecond (target ?h))))))
```

Topos Structure

SET.TOP

Classes and their functions satisfy the axioms for an elementary topos.

- For any two classes X and Y the *exponent* or *hom-class* from X to Y, denoted by $Y^X = \mathsf{SET}[X, Y]$, is the collection of all functions with source X and target Y. There is a binary CNG '`exponent`' function that maps a pair of classes to its associated exponent.

```
(1) (CNG$function exponent)
    (CNG$signature exponent SET$class SET$class SET$class)
    (forall (?c1 ?c2 (SET$class ?c1) (SET$class ?c2) ?f (SET.FTN$function ?f))
       (<=> ((exponent ?c1 ?c2) ?f)
            (and (= (SET.FTN$source ?f) ?c1)
                 (= (SET.FTN$target ?f) ?c2))))
```

- For a fixed class A and any class B, the B-th component of *A-evaluation* $\varepsilon_A(B) : B^A \times A \to B$ maps a pair, consisting of a function $f : A \to B$ and an element $x \in A$ of its source class A, to the image $f(x) \in B$. This is a specific evaluation operator. The KIF term '`evaluation`' represents evaluation.

```
(2) (KIF$function evaluation)
    (KIF$signature evaluation SET$class CNG$function)
    (forall (?a (SET$class ?a))
       (CNG$signature (evaluation ?a) SET$class SET.FTN$function)
    (forall (?a (SET$class ?a) ?b (SET$class ?b))
       (and (= (SET.FTN$source ((evaluation ?a) ?b))
               (SET$binary-product (exponent ?a ?b) ?a))
            (= (SET.FTN$target ((evaluation ?a) ?b)) ?b)))
    (forall (?a (SET$class ?a) ?b (SET$class ?b))
            ?f ((exponent ?a ?b) ?f) ?x (?a ?x))
       (= (((evaluation ?a) ?b) [?f ?x]) (?f ?x))
```

- A finitely complete category K is *Cartesian-closed* when for any object $a \in K$ the product functor $(\text{-}) \times a : K \to K$ has a specified right adjoint $(\text{-})^a : K \to K$ (with a specified counit $\varepsilon_a : (\text{-})^a \times a \Rightarrow Id_K$ called *evaluation*) $(\text{-}) \times a \dashv (\text{-})^a$. Here we present axioms that make the finitely complete quasi-category of classes and functions Cartesian closed. The axiom asserts that binary product is left adjoint to exponent with evaluation as counit: for every function $g : C \times A \to B$ there is a unique function $f : C \to B^A$ called the *A-adjoint* of g that satisfies $f \times id_A \cdot \varepsilon_A(B) = g$. This is a specific right adjoint operator. There is a KIF '`adjoint`' function that represents this right adjoint.

```
(3) (KIF$function adjoint)
    (KIF$signature adjoint SET$class CNG$function)
    (forall (?a (SET$class ?a))
       (CNG$signature (adjoint ?a) SET$class SET$class SET.FTN$function)
    (forall (?a (SET$class ?a) ?c (SET$class ?c) ?b (SET$class ?b))
       (and (= (SET.FTN$source ((adjoint ?a) ?c ?b))
               (exponent (SET$binary-product ?c ?a) ?b))
            (= (SET.FTN$target ((adjoint ?a) ?c ?b))
               (exponent ?c (exponent ?a ?b))))
    (forall (?a ?b ?c (SET$class ?a) (SET$class ?b) (SET$class ?c)
            ?g (SET.FTN$function ?g))
       (=> (and (= (SET.FTN$source ?g) (SET$binary-product ?c ?a))
                (= (SET.FTN$target ?g) ?b))
           (= (((adjoint ?a) ?c ?b) ?g)
              (the (?f (SET.FTN$function ?f))
                   (and (= (SET.FTN$source ?f) ?c)
                        (= (SET.FTN$target ?f) (exponent ?a ?b))
                        (= (SET.FTN$composition
                               (SET.FTN$binary-product ?f (SET.FTN$identity ?a))
                               ((evaluation ?a) ?b))
                           ?g))))))
```

- There is a unary KIF *subobject* function that gives the predicates of (injections on) a class.

```
(1) (KIF$function subobject)
    (KIF$signature subobject SET$class SET$class)
    (forall (?c (SET$class ?c) ?f)
       (<=> ((subobject ?c) ?f)
```

```
            (and (SET.FTN$injection ?f)
                 (= (SET.FTN$target ?f) ?c))))
```

- For any class C an *element* of C is a function $f : 1 \to C$. We can prove the fact that the quasi-topos SET (of classes and their functions) is well-pointed – 1 is a generator; that is, that functions are determined by their effect on their source elements. There is a bijective SET function $el2ftn_C : C \to C^1 = \mathsf{SET}[1, C]$, from ordinary elements of C to (function) elements of C.

```
(2) (CNG$function element)
    (CNG$signature element SET$class SET$class)
    (forall (?c (SET$class ?c))
        (= (element ?c) (exponent SET.LIM$unit ?c)))

    (forall (?f (SET.FTN$function ?f) ?g (SET.FTN$function ?g))
        (=> (and (SET.FTN$parallel-pair [?f ?g])
                 (forall (?h ((element (SET.FTN$source ?f)) ?h))
                     (= (SET.FTN$composition ?h ?f) (SET.FTN$composition ?h ?g)))
            (= ?f ?g)))

(3) (CNG$function el2ftn)
    (CNG$signature el2ftn SET$class SET.FTN$function)
    (forall (?c (SET$class ?c))
        (and (= (SET.FTN$source (el2ftn ?c) ?c)
             (= (SET.FTN$target (el2ftn ?c) (element ?c))
             (forall (?x (?c ?x))
                 (= (((el2ftn ?c) ?x) 0) ?x))))
```

- We can prove the theorem that for any class C the '`(el2ftn ?c)`' function is bijective.

```
    (forall (?c (SET$class ?c))
        (SET.FTN$bijection (el2ftn ?c))
```

- Constant functions are sometimes useful. For any two classes A and B, thought of as source and target classes respectively, there is a binary CNG function '`constant`' that maps elements of the target (codomain) class B to the associated constant function. The constant functions can also be defined as the composition of the '`(unique ?a)`' function from A to 1 and the '`(el2ftn ?b)`' function from 1 to B. that maps an element '`(?b ?y)`' to the associated function.

```
(4) (CNG$function constant)
     (CNG$signature constant SET$class SET$class SET.FTN$function)
     (forall (?a ?b (SET$class ?a) (SET$class ?b))
        (and (= (SET.FTN$source (constant ?a ?b)) ?b)
             (= (SET.FTN$target (constant ?a ?b)) (exponent ?a ?b))))
     (forall (?a ?b (SET$class ?a) (SET$class ?b)
             ?x ?y (?a ?x) (?b ?y))
        (= (((constant ?a ?b) ?y) ?x) ?y))
```

- There is a special class $2 = \{0, 1\}$ called the *truth class* that contains two elements called truth values, where 0 denotes false and 1 denotes true.

```
(5) (SET$class truth)
    (truth 0)
    (truth 1)
    (forall (?x (truth ?x))
       (or (= ?x 0) (= ?x 1)))
```

- There is a special truth element $true : 1 \to 2$ that maps the single element 0 to true (1).

```
(6) (SET.FTN$function true)
    (= (SET.FTN$source true) SET.LIM$unit)
    (= (SET.FTN$target true) SET.LIM$truth)
    (= (true 0) 1)
    (= true ((el2ftn truth) 1))
```

- For any class C there is a *character* function $\chi_C : sub(C) \to 2^C$ that maps subobjects to their characteristic functions.

```
(7) (CNG$function character)
    (CNG$signature character SET$class SET.FTN$function)
    (forall (?c (SET$class ?c))
```

```
      (and (= (SET.FTN$source (character ?c)) (subobject ?c))
           (= (SET.FTN$target (character ?c)) (exponent ?c truth))))
   (forall (?b (SET$class ?b)
            ?f ((SET$subobject ?b) ?f))
       (= ((character ?b) ?f)
          (the (?u (SET.FTN$function ?u))
              (exists (?s (SET.LIM.PBK$opspan ?s))
                  (and (= (true (SET.LIM.PBK$opfirst ?s))
                          (= (?u (SET.LIM.PBK$opsecond ?s))
                          (= (SET.LIM$unique (SET.FTN$source ?f))
                             (SET.LIM.PBK$projection1 ?s))
                          (= ?f (SET.LIM.PBK$projection2 ?s)))))))
```

- The natural numbers $\aleph = \{0, 1, \ldots\}$ is one example of an infinite class. The natural numbers class comes equipped with a *zero* (function) *element* $0 : 1 \to \aleph$ and a *successor function* $\sigma : \aleph \to \aleph$. Moreover, the triple $\langle \aleph, 0, \sigma \rangle$ satisfies the axioms for an *initial algebra* for the endofunctor $1 + (-)$ on the classes and functions. Note that an algebra $\langle S, s_0 : 1 \to S , s : S \to S \rangle$ for the endofunctor $1 + (-)$ and its unique function $h : \aleph \to S$ corresponds to a sequence in the Basic KIF Ontology with the n-th term in the sequence given by $h(n)$.

```
(8) (SET$class natural-numbers)
    ((element natural-numbers) zero)
    ((SET.CLSR$exponent natural-numbers natural-numbers) successor)

    (forall (?c (SET$class ?c)
             ?x ((element ?c) ?x)
             ?f ((SET.FTN$exponent ?c ?c) ?f))
        (exists-unique (?h (SET.FTN$function ?h))
             (and (= (SET.FTN$source ?h) natural-numbers)
                  (= (SET.FTN$target ?h) ?c)
                  (= (SET.FTN$composition zero ?h) ?x)
                  (= (SET.FTN$composition successor ?h)
                     (SET.FTN$composition ?h ?f)))))
```

- We assume the *axiom of extensionality* for functions: if a parallel pair of functions has identical composition on all elements of the source then the two functions are equal.

```
(9) (forall (?f (function ?f) ?g (function ?s))
        (=> (and (= (SET.FTN$source ?f) (SET.FTN$source ?g))
                 (= (SET.FTN$target ?f) (SET.FTN$target ?g))
                 (forall (?x ((element (source ?f)) ?x))
                     (= (SET.FTN$composition ?x ?f)
                        (SET.FTN$composition ?x ?g))))
        (= ?f ?g)))
```

- We assume the *axiom of choice*: any epimorphism has a left inverse (in diagrammatic order).

```
(10) (forall (?f (epimorphism ?f))
         (exists (?g (function ?g))
             (= (composition ?g ?f) (identity (target ?f))))))
```

The Namespace of Large Relations

This namespace will represent large binary relations and their morphisms. More needs to be done here. Some of the terms introduced in this namespace are listed in Table 1.

Table 1: Terms introduced into the large relation namespace

REL	'relation'	'class1', 'class2', 'extent' 'opposite'	'subrelation' 'composition'
REL .ENDO	'endorelation'	'class', 'extent', 'opposite', 'identity' 'reflexive', 'symmetric', 'antisymmetric', 'transitive' 'equivalence-relation', 'equivalence-class', 'quotient', 'canon', 'equivalence-closure' 'order-relation'	'subendorelation'

Relations

REL

A (large) binary relation (Figure 1) is a special case of a conglomerate binary relation with classes for its two coordinates. A class relation is also known as a SET relation. A SET relation is intended to be an abstract semantic notion. Syntactically however, every relation is represented as a binary KIF relation. The signature of SET relations, considered to be CNG relations, is given by their two classes. A SET relation with class X_1 and class X_2 is a triple $\boldsymbol{R} = (X_1, X_2, R)$, where the class $R \subseteq X \times Y$ is the underlying *extent* of the relation.

$$R \subseteq X_1 \times X_2$$

Figure 1: Large Binary Relation

For SET relations both (horizontal) composition and identities are defined. Horizontal composition and identity make the collections of classes and relations into a quasi-category. There is also the notion of relation morphism, which makes this into a quasi-double-category.

- Let 'relation' be the SET namespace term that denotes the *Binary Relation* collection. A binary relation is determined by the triple of its class and extent.

```
(1) (CNG$conglomerate relation)
    (forall (?r (relation ?r)) (CNG$relation ?r))

(2) (CNG$function class1)
    (CNG$signature class1 relation SET$class)

(3) (CNG$function class2)
    (CNG$signature class2 relation SET$class)

    (forall (?r (relation ?r))
        (CNG$signature ?r (class1 ?r) (class2 ?r)))

(4) (CNG$function extent)
    (CNG$signature extent relation SET$class)
    (forall (?r) (relation ?r))
        (SET$subclass
            (extent ?r)
            (SET.LIM.PRD$binary-product (class1 ?r) (class2 ?r))))

    (forall (?r (relation ?r)
             ?x1 ((class1 ?r) ?x1)
             ?x2 ((class2 ?r) ?x2))
         (<=> ((extent ?r) [?x1 ?x2])
              (?r ?x1 ?x2)))

    (forall (?r (relation ?r)
             ?s (relation ?s))
        (=> (and (= (class1 ?r) (class1 ?s))
                 (= (class2 ?r) (class2 ?s))
                 (= (extent ?r) (extent ?s)))
```

```
            (= r s)))
```

- The is a *subrelation* relation. This can be used as a restriction for large binary relations.

```
(5) (CNG$relation subrelation)
    (CNG$signature subrelation relation CNG$relation)
    (forall (?r1 (relation ?r1) ?r2 (CNG$relation ?r2))
        (<=> (subrelation ?r1 ?r2)
             (and (SET$subcollection (class1 ?r1) (CNG$conglomerate1 ?r2))
                  (SET$subcollection (class2 ?r1) (CNG$conglomerate2 ?r2))
                  (SET$subcollection (extent ?r1) (CNG$extent ?r2)))))
```

  - To each relation R, there is an *opposite relation* R^{op}. The classes of R^{op} are the classes of R in reverse order, and the extent of R^{op} is the transpose of the extent of R. The axioms below specify the opposite relation.

```
(6) (CNG$function opposite)
    (CNG$signature opposite relation relation)
    (forall (?r (relation ?r))
        (and (= (class1 (opposite ?r)) (class2 ?r))
             (= (class2 (opposite ?r)) (class1 ?r))
             (forall (?x1 ((class1 ?r) ?x1)
                      ?x2 ((class2 ?r) ?x2))
                 (<=> ((extent (opposite ?r)) [?x2 ?x1])
                      ((extent ?r) [?x1 ?x2]))))))
```

  - An immediate theorem is that the opposite of the opposite is the original relation.

```
(forall (?r (relation ?r))
        (= (opposite (opposite ?r)) ?r))
```

- Two relations R and S are *composable* when the second class of R is the same as the first class of S. There is an binary CNG function *composition* that takes two composable relations and returns their composition.

```
(7) (CNG$function composition)
    (CNG$signature composition relation relation relation)
    (forall (?r (relation ?r) ?s (relation ?s))
        (<=> (exists (?t) (= (composition ?r ?s) ?t))
             (= (class2 ?r) (class1 ?s)))))
    (forall (?r (relation ?r) ?s (relation ?s))
        (=> (= (class2 ?r) (class1 ?s))
            (and (= (class1 (composition ?r ?s)) (class1 ?r))
                 (= (class2 (composition ?r ?s)) (class2 ?s)))))
    (forall (?r (relation ?r) ?s (relation ?s))
        (=> (= (class2 ?r) (class1 ?s))
            (forall (?x ((class1 ?r) ?x) ?z ((class2 ?s) ?z))
                (<=> ((extent (composition ?r ?s)) [?x ?z])
                     (exists (?y ((class2 ?r) ?y))
                         (and ((extent ?r) [?x ?y]) ((extent ?s) [?y ?z])))))))
```

Endorelations

REL.ENDO

- Endorelations are special relations.

```
(1) (CNG$conglomerate endorelation)
    (CNG$subconglomerate endorelation relation)

(2) (CNG$function class)
    (CNG$signature class endorelation SET$class)
    (forall (?r) (endorelation ?r))
        (and (= (class ?r) (REL$class1 ?r))
             (= (class ?r) (REL$class2 ?r))))

(3) (CNG$function extent)
    (CNG$signature extent endorelation SET$class)
    (forall (?r) (endorelation ?r))
        (= (extent ?r) (REL$extent ?r)))
```

- The is a *subendorelation* relation.

```
(4) (CNG$relation subendorelation)
    (CNG$signature subendorelation endorelation endorelation)
    (forall (?r1 (endorelation ?r1) ?r2 (endorelation ?r2))
        (<=> (subendorelation ?r1 ?r2)
             (REL$subrelation ?r1 ?r2)))
```

- To each endorelation R, there is an *opposite endorelation* R^{op}. The class of R^{op} is the class of R, and the extent of R^{op} is the transpose of the extent of R. The axioms below specify the opposite endorelation.

```
(5) (CNG$function opposite)
    (CNG$signature opposite endorelation endorelation)
    (forall (?r (endorelation ?r))
        (and (= (class (opposite ?r)) (class ?r))
             (forall (?x1 ((class ?r) ?x1)
                      ?x2 ((class ?r) ?x2))
                 (<=> ((extent (opposite ?r)) [?x2 ?x1])
                      ((extent ?r) [?x1 ?x2]))))))
```

- An immediate theorem is that the opposite of the opposite is the original endorelation.

```
(forall (?r (endorelation ?r))
        (= (opposite (opposite ?r)) ?r))
```

- For any class A there is an identity endorelation $identity_A$.

```
(6) (CNG$function identity)
    (CNG$signature identity class endorelation)
    (forall (?c (class ?c))
        (= (class (identity ?c)) ?c))
    (forall (?c (class ?c)
             ?x1 (?c ?x1) ?x2 (?c ?x2))
        (<=> ((extent (identity ?c)) [?x1 ?x2])
             (= ?x1 ?x2)))
```

- An endorelation R is *reflexive* when it contains the identity relation.

```
(7) (CNG$conglomerate reflexive)
    (CNG$subconglomerate reflexive endorelation)
    (forall (?r (endorelation ?r))
        (<=> (reflexive ?r)
             (forall (?x ((class ?r) ?x))
                 ((extent ?r) [?x ?x])))))
```

- An endorelation R is *symmetric* when it contains the opposite relation.

```
(8) (CNG$conglomerate symmetric)
    (CNG$subconglomerate symmetric endorelation)
    (forall (?r (endorelation ?r))
        (<=> (symmetric ?r)
             (forall (?x1 ((class ?r) ?x1) ?x2 ((class ?r) ?x2))
                 (=> ((extent ?r) [?x1 ?x2])
                     ((extent ?r) [?x2 ?x1]))))))
```

- An endorelation R is *antisymmetric* when it contains the intersection of the relation with its opposite is contained in the identity relation.

```
(9) (CNG$conglomerate antisymmetric)
    (CNG$subconglomerate antisymmetric endorelation)
    (forall (?r (endorelation ?r))
        (<=> (antisymmetric ?r)
             (forall (?x1 ((class ?r) ?x1) ?x2 ((class ?r) ?x2))
                 (=> (and ((extent ?r) [?x1 ?x2])
                          ((extent ?r) [?x2 ?x1]))
                     (= ?x1 ?x2))))))
```

- An endorelation R is *transitive* when it contains the composition with itself.

```
(10) (CNG$conglomerate transitive)
     (CNG$subconglomerate transitive endorelation)
     (forall (?r (endorelation ?r))
         (<=> (transitive ?r)
              (forall (?x1 ((class ?r) ?x1)
                       ?x2 ((class ?r) ?x2)
```

```
                ?x3 ((class ?r) ?x3))
            (=> (and ((extent ?r) [?x1 ?x2])
                     ((extent ?r) [?x2 ?x3]))
                ((extent ?r) [?x1 ?x3])))))
```

- An *equivalence relation E* is a reflexive, symmetric and transitive endorelation. An equivalence relation determines a *quotient* class and a *canon*(ical) surjection. Every endorelation generates an equivalence-relation, the smallest containing it.

```
(11) (CNG$conglomerate equivalence-relation)
     (CNG$subconglomerate equivalence-relation endorelation)
     (forall (?e (endorelation ?e))
         (<=> (equivalence-relation ?e)
              (and (reflexive ?e) (symmetric ?e) (transitive ?e))))

(12) (KIF$function equivalence-class)
     (KIF$signature equivalence-class equivalence-relation CNG$function)
     (forall (?e (equivalence-relation ?e))
         (and (CNG$signature (equivalence-class ?e) (class ?e) SET$class))
              (forall (?x1 ((class ?e) ?x))
                      ?x2 ((class ?e) ?x2))
                  (<=> (((equivalence-class ?e) ?x1) ?x2)
                       ((extent ?e) [?x1 ?x2])))))

(13) (CNG$function quotient)
     (CNG$signature quotient equivalence-relation class)
     (forall (?e (equivalence-relation ?e)
              ?c (SET$class ?c))
         (<=> ((quotient ?e) ?c)
              (exists (?x ((class ?e) ?x))
                  (= ?c ((equivalence-class ?e) ?x)))))

(14) (CNG$function canon)
     (CNG$signature canon equivalence-relation SET.FTN$surjection)
     (forall (?e (equivalence-relation ?e)
         (and (= (SET.FTN$source (canon ?e)) (class ?e))
              (= (SET.FTN$target (canon ?e)) (quotient ?e))))
              (= ((canon ?e) ?x) ((equivalence-class ?e) ?x))))

(15) (CNG$function equivalence-closure)
     (CNG$signature equivalence-closure endorelation equivalence-relation)
     (forall (?r (endorelation ?r))
         (= (equivalence-closure ?r)
            (the (?e (equivalence-relation ?e))
                (and (subendorelation ?r ?e)
                     (forall (?e1 (equivalence-relation ?e1))
                         (=> (subendorelation ?r ?e1)
                             (subendorelation ?e ?e1)))))))
```

- An *order relation E* is a reflexive, antisymmetric and transitive endorelation.

```
(16) (CNG$conglomerate order-relation)
     (CNG$subconglomerate order-relation endorelation)
     (forall (?r (endorelation ?r))
         (<=> (order-relation ?r)
              (and (reflexive ?r) (antisymmetric ?r) (transitive ?r))))
```

The Namespace of Large Orders

This namespace will represent large orders and their monotonic functions. Some of the terms introduced in this namespace are listed in Table 1. This only scratches the surface of this namespace.

Table 1: Terms introduced into the large order namespace

ORD	'order'	'order', 'class', 'relation'	

Orders

ORD

- An *order* $\boldsymbol{A} = \langle A, \leq \rangle$ is a pair consisting of a class A and an order-relation $\leq \subseteq A \times A$ on that class.

```
(1) (CNG$conglomerate order)

(2) (CNG$function class)
    (CNG$signature class order SET$class)

(3) (CNG$function relation)
    (CNG$signature relation order REL.ENDO$order-relation)

    (forall (?o (order ?o))
        (= (REL.ENDO$class (relation ?o)) (class ?o)))
```

- For any class A there is an identity order $identity_A$.

```
(6) (CNG$function identity)
    (CNG$signature identity class order)
    (forall (?c (class ?c))
        (and (= (class (identity ?c)) ?c)
             (= (relation (identity ?c)) (REL$identity ?c))))
```

The Category Theory Ontology

As listed in Table 1, the namespaces in the Category Theory Ontology import and use terms from the core namespace of classes and their functions.

Table 1: Terms imported and used from other namespaces

SET	‘conglomerate’, ‘class’, ‘subclass’ ‘function’, ‘signature’, ‘source’, ‘target’ ‘composition’, ‘identity’, ‘inclusion’ ‘terminal’, ‘unique’ ‘opspan’, ‘opvertex’, ‘opfirst’, ‘opsecond’ ‘pullback’, ‘pullback-projection1’, ‘pullback-projection2’

Table 2 lists the terms defined and axiomatized in the namespaces of the Category Theory Ontology. The core terminology is listed in boldface.

Table 2: Terms introduced in the Category Theory Ontology

	CNG$conglomerate	Unary CNG$function	Binary/Ternary CNG$function
GPH	‘**graph**’ ‘**small**’	‘**object**’, ‘**morphism**’, ‘**source**’, ‘**target**’ ‘**opposite**’ ‘**unit**’	‘multiplication-opspan’, ‘**multiplication**’
GPH.MOR	‘**graph-morphism**’ ‘**2-cell**’	‘**source**’, ‘**target**’, ‘**object**’, ‘**morphism**’ ‘**opposite**’ ‘**unit**’ ‘**identity**’ ‘**isomorphism**’, ‘**inverse**’, ‘**isomorphic**’ ‘**tau**’ ‘**left**’, ‘**right**’	‘multiplication-cone’, ‘**multiplication**’ ‘**composition**’ ‘first-cone’, ‘opspan12-3’, ‘second-cone’ ‘**alpha**’, ‘**associativity**’
CAT	‘**category**’ ‘**small**’ ‘**monomorphism**’, ‘**epimorphism**’, ‘**isomorphism**’	‘**underlying**’, ‘**mu**’, ‘**eta**’ ‘**composition**’, ‘**identity**’ ‘**object**’, ‘**morphism**’, ‘**source**’, ‘**target**’ ‘composable-opspan’, ‘**composable**’, ‘first’, ‘second’, ‘**opposite**’ ‘object-pair’, ‘object-binary-product’ ‘source-target’, ‘hom-object’, ‘parallel-pair’ ‘left-composable’, ‘left-composition’ ‘right-composable’, ‘right-composition’	
FUNC	‘**functor**’	‘**source**’, ‘**target**’, ‘**underlying**’ ‘**object**’, ‘**morphism**’ ‘**unique**’, ‘**element**’, ‘**opposite**’ ‘**identity**’, ‘**diagonal**’	‘**composition**’
NAT	‘**natural-transformation**’	‘**source-functor**’, ‘**target-functor**’, ‘**source-category**’, ‘**target-category**’, ‘**component**’ ‘**vertical-identity**’ ‘**horizontal-identity**’	‘**vertical-composition**’ ‘**horizontal-composition**’
ADJ	‘**adjunction**’	‘**underlying-category**’, ‘**free-category**’, ‘**left-adjoint**’, ‘**right-adjoint**’, ‘**unit**’, ‘**counit**’ ‘**adjunction-monad**’ ‘**free**’, ‘**eilenberg-moore-comparison**’, ‘**extension**’, ‘**kliesli-comparison**’ ‘**reflection**’, ‘**coreflection**’	

ADJ .MOR	‘**conjugate-pair**	‘**source**’, ‘**target**’, ‘**left-conjugate**’, ‘**right-conjugate**’	
MND	‘**monad**’	‘**underlying-category**’, ‘**underlying-functor**’, ‘**unit**’, ‘**multiplication**’	
	‘**monad-morphism**’	‘**source**’, ‘**target**’, ‘**underlying-natural-transformation**’	
MON .ALG	‘**algebra**’	‘**underlying-object**’, ‘**structure-map**’	
	‘**homomorphism**’	‘**source**’, ‘**target**’, ‘**underlying-morphism**’, ‘composable-opspan’, ‘composable’, ‘**composition**’, ‘**identity**’	
	‘**eilenberg-moore**’	‘**underlying-eilenberg-moore**’, ‘**free-eilenberg-moore**’, ‘**unit-eilenberg-moore**’, ‘**counit-eilenberg-moore**’, ‘**adjunction-eilenberg-moore**’	
	‘**kliesli**’	‘kliesli-morphism-opspan’, ‘kliesli-identity-cocone’, ‘kliesli-composable-opspan’, ‘extension’, ‘**underlying-kliesli**’, ‘embed’, ‘**free-kliesli**’, ‘**unit-kliesli**’, ‘**counit-kliesli**’, ‘**adjunction-kliesli**’	
COL		‘**initial**’, ‘**counique**’ ‘**diagram**’, ‘**shape** ‘**cocone**’, ‘**cocone-diagram**’, ‘**base**’, ‘**opvertex**’ ‘**colimiting-cocone**’, ‘**colimit**’ ‘**comediator**’ ‘**cocomplete**’, ‘**small-cocomplete**’	
COL .COPRD		‘**diagram**’, ‘**shape** ‘**cocone**’, ‘**cocone-diagram**’, ‘**base**’, ‘**opvertex**’ ‘**colimiting-cocone**’, ‘**binary-coproduct**’, ‘**injection1**’, ‘**injection2**’ ‘**comediator**’	
COL .PSH		‘**diagram**’, ‘**shape** ‘**cocone**’, ‘**cocone-diagram**’, ‘**base**’, ‘**opvertex**’ ‘**colimiting-cocone**’ ‘**pushout**’, ‘**injection1**’, ‘**injection2**’ ‘**comediator**’	

The Namespace of Large Graphs

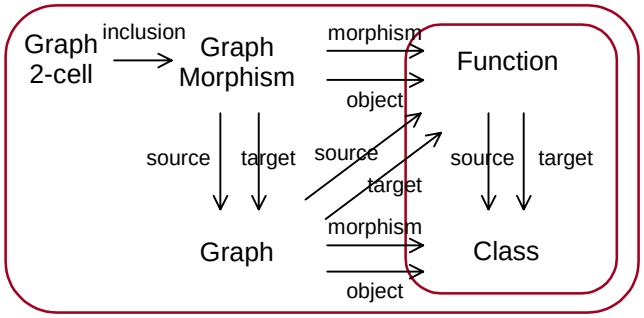
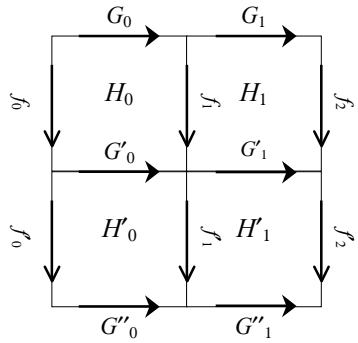


Diagram 1: Core Conglomerates and Functions

Table 1: Elements of the 2-dimensional category GRAPH

square	=	graph morphism
vertical category	=	**GRAPH** (**Set**)
vertical morphism	=	function
vertical composition	=	graph morphism (function) composition
vertical identity	=	graph morphism (function) identity
horizontal pseudo-category		
horizontal morphism	=	graph
horizontal composition	=	graph morphism (graph) multiplication
horizontal identity	=	graph morphism (graph) unit



Graphs

GPH

- A (*large*) *graph* G (Figure 1) consists of a class $obj(G)$ of objects, a class $mor(G)$ of morphisms (arrows), a *source* (domain) function $src(G) : mor(G) \to obj(G)$ and a *target* (codomain) function $tgt(G) : mor(G) \to obj(G)$. A morphism $m \in mor(G)$, with source object $src(G)(m) = o_0 \in obj(G)$ and target object $tgt(G)(m) = o_1 \in obj(G)$, is usually represented graphically with the notation $m : o_0 \to o_1$.

$$src(G)$$
$$mor(G) \rightrightarrows obj(G)$$
$$tgt(G)$$

Figure 1: Graph

The following is a IFF representation for the elements of a graph. Axiom (1) defines the graph conglomerate. Axioms (2–4) model the graph structure of Figure 7. In axiom (2) and axiom (3), the CNG functions '`object`' and '`morphism`' are used to specify the objects and morphisms of the graph. These are unary, since they take a graph as a parameter – the terms '`(object ?g)`' and '`(morphism ?g)`' are the actual classes. This parametric technique is used throughout the formulation of IFF. The CNG functions '`source`' and '`target`' in axioms (4) and (5) represent the source and target functions that assign objects to morphisms. Graphs are uniquely determined by their (object, morphism, source, target) quadruple.

```
(1) (CNG$conglomerate graph)

(2) (CNG$function object)
    (CNG$signature object graph SET$class)

(3) (CNG$function morphism)
```

```
    (CNG$signature morphism graph SET$class)

(4) (CNG$function source)
    (CNG$signature source graph SET.FTN$function)
    (forall (?g (graph ?g))
        (and (= (SET.FTN$source (source ?g)) (morphism ?g))
             (= (SET.FTN$target (source ?g)) (object ?g))))

(5) (CNG$function target)
    (CNG$signature target graph SET.FTN$function)
    (forall (?g (graph ?g))
        (and (= (SET.FTN$source (target ?g)) (morphism ?g))
             (= (SET.FTN$target (target ?g)) (object ?g))))

    (forall (?g1 (graph ?g1) ?g2 (graph ?g2))
        (=> (and (= (object ?g1) (object ?g2))
                 (= (morphism ?g1) (morphism ?g2))
                 (= (source ?g1) (source ?g2))
                 (= (target ?g1) (target ?g2)))
            (= ?g1 ?g2)))
```

- To each graph G, there is an *opposite graph* G^{op}. The opposite graph is also called the *dual graph*. The objects of G^{op} are the objects of G, and the morphisms of G^{op} are the morphisms of G. However, the source and target functions are reversed: $src(G^{op}) = tgt(G)$ and $tgt(G^{op}) = src(G)$. The type restriction axiom (6) specifies the notion of the opposite graph.

```
(6) (CNG$function opposite)
    (CNG$signature opposite graph graph)
    (forall (?g (graph ?g))
        (and (= (object (opposite ?g)) (object ?g))
             (= (morphism (opposite ?g)) (morphism ?g))
             (= (source (opposite ?g)) (target ?g))
             (= (target (opposite ?g)) (source ?g))))
```

  An immediate theorem is that the opposite of the opposite is the original graph.

```
    (forall (?g (graph ?g))
        (= (opposite (opposite ?g)) ?g))
```

- A graph G is small when both the object and morphism classes are (small) sets.

```
(7) (CNG$conglomerate small)
    (CNG$subconglomerate small graph)
    (forall (?g (graph ?g))
        (<=> (small ?g)
             (and (set$set (object ?g))
                  (set$set (morphism ?g)))))
```

Multiplication

- Two graphs G_0 and G_1 are *composable* when they share a class of objects $obj(G_0) = obj = obj(G_1)$. For two composable graphs there is an associated *multiplication* graph $G_0 \otimes G_1$ (Figure 2), whose class of morphisms is the class of *composable pairs* of morphisms defined above. This is the pullback in foundations along the target function $tgt(G_0) : mor(G_0) \to obj$ and the source function $src(G_1) : mor(G_1) \to obj$.

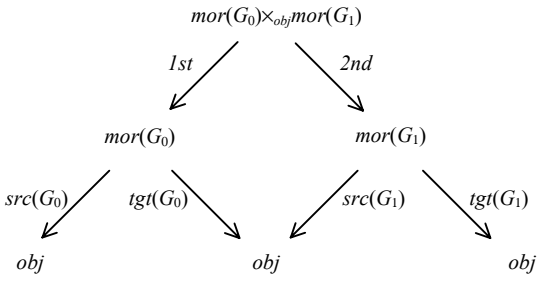


Figure 2: Multiplication Graph

$mor(G_0) \times_{Obj} mor(G_1) =$
$\{(m_0, m_1) \mid m_0 \in mor(G_0) \text{ and } m_1 \in mor(G_1) \text{ and } tgt(R_0)(m_0) = src(R_1)(m_1)\}$

Here is the KIF formalism. Each composable pair of graphs has an auxiliary foundational opspan represented as the IFF term '`(multiplication-opspan ?g0 ?g1)`' and axiomatized in (8). This opspan

allows us to refer to the appropriate foundational pullback. The multiplication graph $G_0 \otimes G_1$ is represented as the IFF term '`(multiplication ?g0 ?g1)`' and axiomatized in (9).

```
(8) (CNG$function multiplication-opspan)
    (CNG$signature multiplication-opspan graph graph SET.LIM.PBK$opspan)
    (forall (?g0 (graph ?g0) ?g1 (graph ?g1))
          (<=> (exists (?s (SET.LIM.PBK$opspan ?s))
                    (= (multiplication-opspan ?g0 ?g1) ?s))
               (= (object ?g0) (object ?g1))))
    (forall (?g0 (graph ?g0) ?g1 (graph ?g1))
          (=> (= (object ?g0) (object ?g1))
              (and (= (SET.LIM.PBK$opvertex (multiplication-opspan ?g0 ?g1))
                      (object ?g0))
                   (= (SET.LIM.PBK$opfirst (multiplication-opspan ?g0 ?g1))
                      (target ?g0))
                   (= (SET.LIM.PBK$opsecond (multiplication-opspan ?g0 ?g1))
                      (source ?g1))))))

(9) (CNG$function multiplication)
    (CNG$signature multiplication graph graph graph)
    (forall (?g0 (graph ?g0) ?g1 (graph ?g1))
        (<=> (exists (?g (graph ?g))
                  (= (multiplication ?g0 ?g1) ?g))
             (= (object ?g0) (object ?g1))))
    (forall (?g0 (graph ?g0) ?g1 (graph ?g1))
        (=> (= (object ?g0) (object ?g1))
            (and (= (object (multiplication ?g0 ?g1))
                    (object ?g0))
                 (= (morphism (multiplication ?g0 ?g1))
                    (SET.LIM.PBK$pullback (multiplication-opspan ?g0 ?g1)))
                 (= (source (multiplication ?g0 ?g1))
                    (SET$composition
                        (SET.LIM.PBK$projection1 (multiplication-opspan ?g0 ?g1))
                        (source ?g0)))
                 (= (target (multiplication ?g0 ?g1))
                    (SET$composition
                        (SET.LIM.PBK$projection2 (multiplication-opspan ?g0 ?g1))
                        (target ?g1))))))
```

Unit

- For any class (of objects) C there is a *unit* graph 1_C (Figure 3) whose classes of objects and morphisms are C, and whose source and target functions is the SET identity function on C. The unit graph has the following formalization.

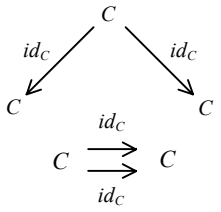


Figure 3: Unit Graph

```
(10) (CNG$function unit)
     (CNG$signature unit SET$class graph)
     (forall (?c (SET$class ?c))
         (and (= (object (unit ?c)) ?c)
              (= (morphism (unit ?c)) ?c)
              (= (source (unit ?c)) (SET.FTN$identity ?c))
              (= (target (unit ?c)) (SET.FTN$identity ?c))))
```

An immediate theorem is that the opposite of the unit graph is itself.

```
(forall (?c (SET$class ?c))
    (= (opposite (unit ?c)) (unit ?c)))
```

Graph Morphisms

`GPH.MOR`

- A *graph morphism* (Figure 4) $H : G \Rightarrow G'$ from graph G to graph G' consists of two functions, an *object function* and a *morphism function*, that preserve source and target (the diagram in Figure 4 is commutative). The object function $obj(H) : Obj(G) \rightarrow Obj(G')$ assigns to each object of G an object of G', and the morphism function $mor(H) : Mor(G) \rightarrow Mor(G')$ assigns to each morphism of G a mor-

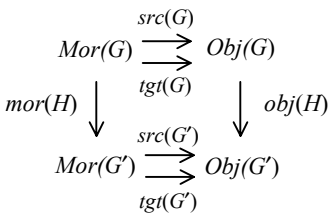


Figure 4: Graph Morphism

phism of G'. In the graph morphism that is presented in Figure 4, the diagram is asserted to be commutative. What this means is that the component functions must preserve source and target in the sense that the following constraints must be satisfied.

$mor(H) \cdot src(G') = src(G) \cdot obj(H)$ and $mor(H) \cdot tgt(G') = tgt(G) \cdot obj(H)$

The following is a formalization of a graph morphism. The CNG functions '`source`' and '`target`' represent source (domain) and target (codomain) operations that assign graphs to graph morphisms. The SET function '`(object ?h)`' specifies the object function of a graph morphism '`?h`', and the SET function '`(morphism ?h)`' specifies the morphism function of the graph morphism. The CNG functions '`object`' and '`morphism`' have the graph morphism as a parameter. The last axiom represents preservation of source and target. Graph morphisms are uniquely determined by their (source, target, object, morphism) quadruple.

```
(1) (CNG$conglomerate graph-morphism)

(2) (CNG$function source)
    (CNG$signature source graph-morphism GPH$graph)

(3) (CNG$function target)
    (CNG$signature target graph-morphism GPH$graph)

(4) (CNG$function object)
    (CNG$signature object graph-morphism SET.FTN$function)
    (forall (?h (graph-morphism ?h))
        (and (= (SET.FTN$source (object ?h)) (GPH$object (source ?h)))
             (= (SET.FTN$target (object ?h)) (GPH$object (target ?h)))))

(5) (CNG$function morphism)
    (CNG$signature morphism graph-morphism SET.FTN$function)
    (forall (?h (graph-morphism ?h))
        (and (= (SET.FTN$source (morphism ?h)) (GPH$morphism (source ?h)))
             (= (SET.FTN$target (morphism ?h)) (GPH$morphism (target ?h)))))

    (forall (?h (graph-morphism ?h))
        (and (= (SET.FTN$composition (morphism ?h) (GPH$source (target ?h)))
                (SET.FTN$composition (GPH$source (source ?h)) (object ?h))))
             (= (SET.FTN$composition (morphism ?h) (GPH$target (target ?h)))
                (SET.FTN$composition (GPH$target (source ?h)) (object ?h))))))

    (forall (?h1 (graph-morphism ?h1) ?h2 (graph-morphism ?h2))
        (=> (and (= (source ?h1) (source ?h2))
                 (= (target ?h1) (target ?h2)))
                 (= (object ?h1) (object ?h2))
                 (= (morphism ?h1) (morphism ?h2))
            (= ?h1 ?h2)))
```

- To each graph morphism $H : G \Rightarrow G'$, there is an *opposite graph morphism* $H^{op} : G^{op} \Rightarrow G'^{op}$. The opposite graph morphism is also called the *dual graph morphism*. The object function of H^{op} is the object function of H, and the morphism function of H^{op} is the morphism function of H. However, the source and target graphs are the opposite: $src(H^{op}) = src(H)^{op}$ and $tgt(H^{op}) = tgt(H)^{op}$. Axiom (6) specifies an opposite graph morphism.

```
(6) (CNG$function opposite)
    (CNG$signature opposite graph-morphism graph-morphism)
    (forall (?h (graph-morphism ?h))
        (and (= (source (opposite ?h)) (GPH$opposite (source ?h)))
             (= (target (opposite ?h)) (GPH$opposite (target ?h)))
             (= (object (opposite ?h)) (object ?h))
             (= (morphism (opposite ?h)) (morphism ?h)))))
```

An immediate theorem is that the opposite of the opposite of a graph morphism is the original graph morphism.

```
    (forall (?h (graph-morphism ?h))
        (= (opposite (opposite ?h)) ?h))
```

Multiplication

- The multiplication operation on graphs can be extended to graph morphisms. For any two graphs morphisms $H_0 : G_0 \Rightarrow G'_0$ and $H_1 : G_1 \Rightarrow G'_1$, which are horizontally composable, in that they share a common object function $obj(H_0) = obj = obj(H_1)$, there is a *multiplication* graph morphism $H_1 \otimes H_2 : G_0 \otimes G_1 \Rightarrow G'_0 \otimes G'_1$, (Figure 5) whose object function is the common object function and whose morphism function is determined by pullback. This is the SET pullback along the morphism functions of H_0 and H_1.

  The multiplication graph morphism $H_0 \otimes H_1$ is represented as the SET term '`(multiplication ?h0 ?h1)`'. The formalism for the multiplication of graphs used an auxiliary associated pullback diagram (opspan). In comparison and contrast, the formalism for the multiplication of graph morphisms uses an auxiliary pullback cone represented as the term '`(multiplication-cone ?h0 ?h1)`' and axiomatized in (7). The multiplication span morphism $H_0 \otimes H_1$ is represented as the term '`(multiplication ?h0 ?h1)`' and axiomatized in (8).

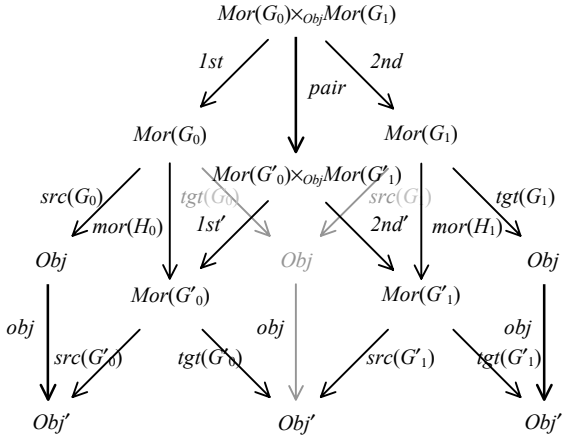


Figure 5: Multiplication Graph Morphism

```
(7) (CNG$function multiplication-cone)
    (CNG$signature multiplication-cone graph-morphism graph-morphism SET.LIM.PBK$cone)
    (forall (?h0 (graph-morphism ?h0) ?h1 (graph-morphism ?h1))
        (<=> (exists (?r (SET.LIM.PBK$cone ?r))
                 (= (multiplication-cone ?h0 ?h1) ?r))
             (= (object ?h0) (object ?h1))))
    (forall (?h0 (graph-morphism ?h0) ?h1 (graph-morphism ?h1))
        (=> (= (object ?h0) (object ?h1))
            (and (= (SET.LIM.PBK$cone-diagram (multiplication-cone ?h0 ?h1))
                    (GPH$multiplication-opspan (target ?h0) (target ?h1)))
                 (= (SET.LIM.PBK$vertex (multiplication-cone ?h0 ?h1))
                    (SET.LIM.PBK$pullback
                        (GPH$multiplication-opspan (source ?h0) (source ?h1))))
                 (= (SET.LIM.PBK$first (multiplication-cone ?h0 ?h1))
                    (SET.FTN$composition
                        (SET.LIM.PBK$projection1
                            (GPH$multiplication-opspan (source ?h0) (source ?h1)))
                        (morphism ?h0)))
                 (= (SET.LIM.PBK$second (multiplication-cone ?h0 ?h1))
                    (SET$composition
                        (SET.LIM.PBK$projection2
                            (GPH$multiplication-opspan (source ?h0) (source ?h1)))
                        (morphism ?h1)))))))

(8) (CNG$function multiplication)
    (CNG$signature multiplication graph-morphism graph-morphism graph-morphism)
    (forall (?h0 (graph-morphism ?h0) ?h1 (graph-morphism ?h1))
        (<=> (exists (?h (graph-morphism ?h))
                 (= (multiplication ?h0 ?h1) ?h))
             (= (object ?h0) (object ?h1))))
    (forall (?h0 (graph-morphism ?h0) ?h1 (graph-morphism ?h1))
        (=> (= (object ?h0) (object ?h1))
            (and (= (source (multiplication ?h0 ?h1))
                    (GPH$multiplication (source ?h0) (source ?h1)))
                 (= (target (multiplication ?h0 ?h1))
                    (GPH$multiplication (target ?h0) (target ?h1)))
                 (= (object (multiplication ?h0 ?h1))
                    (object ?h0))
                 (= (morphism (multiplication ?h0 ?h1))
                    (SET.LIM.PBK$mediator (multiplication-cone ?h0 ?h1)))))))
```

Unit

- For any function (of objects) $f : A \to B$ there is a *unit* graph morphism (Figure 6) $1_f : 1_A \Rightarrow 1_B$, whose source and target graphs are the unit graphs for A and B, and whose object and morphism functions are f. The unit graph morphism has the following representation.

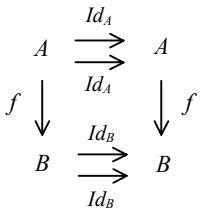


Figure 6: Unit Graph Morphism

```
(9) (CNG$function unit)
    (CNG$signature unit SET.FTN$function graph-morphism)
    (forall (?f (SET.FTN$function ?f))
        (and (= (source (unit ?f)) (GPH$unit (SET.FTN$source ?f)))
             (= (target (unit ?f)) (GPH$unit (SET.FTN$target ?f)))
             (= (object (unit ?f)) ?f)
             (= (morphism (unit ?f)) ?f)))
```

It is clear that the opposite of the unit graph morphism is itself.

```
    (forall (?f (SET.FTN$function ?f))
        (= (opposite (unit ?f)) (unit ?f)))
```

2-Dimensional Category Structure

- A pair of graph morphisms H_1 and H_2 is *composable* when the target graph of H_1 is the source graph of H_2. For any composable pair of graph morphisms $H_1 : G_0 \Rightarrow G_1$ and $H_2 : G_1 \Rightarrow G_2$ there is a *composition graph morphism* $H_1 \circ H_2 : G_0 \Rightarrow G_2$. Its object and morphism functions are the compositions of the object and morphism functions of the component graph morphisms, respectively. For any graph G there is an *identity graph morphism* $id_G : G \Rightarrow G$ on that graph. Its object and morphism functions are the identity functions on the object and morphism classes of that graph, respectively. The following represents the declaration of composition and identity and the equivalence of composability.

```
(10) (CNG$function composition)
     (CNG$signature composition graph-morphism graph-morphism graph-morphism)
      (forall (?h1 (graph-morphism ?h1) ?h2 (graph-morphism ?h2))
         (<=> (exists (?h (graph-morphism ?h)) (= ?h (composition ?h1 ?h2)))
              (= (target ?h1) (source ?h2))))

(11) (CNG$function identity)
     (CNG$signature identity GPH$graph graph-morphism)
```

These graph morphism operations satisfy the following typing constraints.

$$src(id_G) = G = tgt(id_G),\ src(H_1 \circ H_2) = src(H_1),\ tgt(H_1 \circ H_2) = tgt(H_2)$$

for all graphs G and all composable pairs of graph morphisms H_1 and H_2. These theorems are expressed in KIF with the following formalism.

```
(12) (forall (?h1 (graph-morphism ?h1) ?h2 (graph-morphism ?h2))
         (=> (= (target ?h1) (source ?h2))
             (and (= (source (composition ?h1 ?h2)) (source ?h1))
                  (= (target (composition ?h1 ?h2)) (target ?h2)))))

(13) (forall (?g (GPH$graph ?g))
         (and (= (source (identity ?g)) ?g)
              (= (target (identity ?g)) ?g)))
```

- Graph morphism composition and identity are defined componentwise in axioms (14–15). The preservation of source and target functions for composite and identity graph morphisms follows from this as theorems.

```
(14) (forall (?h1 (graph-morphism ?h1) ?h2 (graph-morphism ?h2))
         (=> (= (target ?h1) (source ?h2))
             (and (= (object (composition ?h1 ?h2))
                     (SET.FTN$composition (object ?h1) (object ?h2)))
                  (= (morphism (composition ?h1 ?h2))
                     (SET.FTN$composition (morphism ?h1) (morphism ?h2))))))

(15) (forall (?g (GPH$graph ?g))
         (and (= (object (identity ?g))
```

```
            (SET.FTN$identity (GPH$object ?g)))
        (= (morphism (identity ?g))
           (SET.FTN$identity (GPH$morphism ?g)))))
```

It can be shown that graph morphism composition satisfies the following associative law

$(H_1 \circ H_2) \circ H_3 = H_1 \circ (H_2 \circ H_3)$

for all composable pairs of graph morphisms (H_1, H_2) and (H_2, H_3), and graph morphism identity satisfies the following identity laws

$Id_{G0} \cdot H = H$ and $H = H \cdot Id_{G1}$

for any graph morphism $H : G_0 \Rightarrow G_1$ with source graph G_0 and target graph G_1. Graphs as objects and graph morphisms as morphisms form a quasi-category ("quasi" since this is at the level of conglomerates in foundations). This has the following expression in an external namespace.

```
(forall (?h1 (GPH.MOR$graph-morphism ?h1)
         ?h2 (GPH.MOR$graph-morphism ?h2)
         ?h3 (GPH.MOR$graph-morphism ?h3))
    (=> (and (= (GPH.MOR$target ?h1) (GPH.MOR$source ?h2))
             (= (GPH.MOR$target ?h2) (GPH.MOR$source ?h3)))
        (= (GPH.MOR$composition (GPH.MOR$composition ?h1 ?h2) ?h3)
           (GPH.MOR$composition ?h1 (GPH.MOR$composition ?h2 ?h3)))))

(forall (?h (GPH.MOR$graph-morphism ?h))
    (and (= (GPH.MOR$composition (GPH.MOR$identity (GPH.MOR$source ?h)) ?h) ?h)
         (= (GPH.MOR$composition ?h (GPH.MOR$identity (GPH.MOR$target ?h))) ?h)))
```

- Two oppositely directed graph morphisms $H : G_0 \rightarrow G_1$ and $H' : G_1 \rightarrow G_0$ are *inverses* of each other when $H \circ H' = id_{G0}$ and $H' \circ H = id_{G1}$. *A graph isomorphism* is a graph morphism that has an inverse. With these laws, we can prove the theorem that an inverse to a graph morphism is unique. The *inverse* function maps an isomorphism to its inverse (another isomorphism). This is a bijection. Two graphs are said to be *isomorphic* when there is a graph isomorphism between them.

```
(16) (CNG$conglomerate isomorphism)
     (CNG$subconglomerate isomorphism graph-morphism)
     (forall (?h (graph-morphism ?h))
         (<=> (isomorphism ?h)
              (exists (?h1 (graph-morphism ?h1))
                  (and (= (source ?h1) (target ?h))
                       (= (target ?h1) (source ?h))
                       (= (composition ?h ?h1) (identity (source ?h)))
                       (= (composition ?h1 ?h) (identity (target ?h))))))))

     (forall (?h (graph-morphism ?h)
              ?h1 (graph-morphism ?h1) ?h2 (graph-morphism ?h2))
         (=> (and (= (source ?h1) (target ?h)) (= (source ?h2) (target ?h))
                  (= (target ?h1) (source ?h)) (= (target ?h2) (source ?h))
                  (= (composition ?h ?h1) (identity (source ?h)))
                  (= (composition ?h ?h2) (identity (source ?h)))
                  (= (composition ?h1 ?h) (identity (target ?h)))
                  (= (composition ?h2 ?h) (identity (target ?h))))
             (= ?h1 ?h2)))

(17) (CNG$function inverse)
     (CNG$signature inverse isomorphism isomorphism)
     (forall (?h (isomorphism ?h))
         (= (inverse ?h)
            (the (?h1 (isomorphism ?h1))
                (and (= (source ?h1) (target ?h))
                     (= (target ?h1) (source ?h))
                     (= (composition ?h ?h1) (identity (source ?h)))
                     (= (composition ?h1 ?h) (identity (target ?h))))))))

     (forall (?h (isomorphism ?h))
         (= (inverse (inverse ?h)) ?h))

(18) (CNG$relation isomorphic)
```

```
(CNG$signature isomorphic GPH$graph GPH$graph)
(forall ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
    (<=> (isomorphic ?g1 ?g2)
         (exists (?h) (and (isomorphism ?h)
                           (= (source ?h) ?g1)
                           (= (target ?h) ?g2)))))
```

- A *graph 2-cell* $H : G \to G'$ from graph G to graph G' is a graph morphism whose object function is an identity. This means that the source and target graphs have the same object class $obj(G) = obj = obj(G')$.

```
(19) (CNG$conglomerate 2-cell)
     (CNG$subconglomerate 2-cell graph-morphism)

     (forall (?h (graph-morphism ?h))
         (<=> (2-cell ?h)
              (and (= (GPH$object (source ?h)) (GPH$object (target ?h)))
                   (= (object ?h) (SET.FTN$identity (GPH$object (source ?h)))))))
```

- The opposite of the multiplication of two graphs is isomorphic to the multiplication of the opposites of the component graphs. This isomorphism is mediated by the *tau* or *twist* graph morphism, which is both an isomorphism and a 2-cell.

  $\tau_{G0,\,G1} : G_1^{\,op} \otimes G_0^{\,op} \to (G_0 \otimes G_1)^{op}$.

  The morphism function of tau is the SET tau function '`(tau ?s)`' for the multiplication pullback diagram (opspan) '`?s`'. Axiom (20) is the definition of the tau graph morphism.

```
(20) (CNG$function tau)
     (CNG$signature tau GPH$graph GPH$graph-morphism)
     (forall (?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (=> (= (GPH$object ?g1) (GPH$object ?g2))
             (and (= (source (tau ?g1 ?g2))
                     (GPH$multiplication (GPH$opposite ?g2) (GPH$opposite ?g1)))
                  (= (target (tau ?g1 ?g2))
                     (GPH$opposite (GPH$multiplication ?g1 ?g2)))
                  (= (object (tau ?g1 ?g2))
                     (SET.FTN$identity (GPH$object ?g1)))
                  (= (morphism (tau ?g1 ?g2))
                     (SET.LIM.PBK$tau (GPH$multiplication-opspan ?g1 ?g2))))))

     (forall (?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (=> (= (GPH$object ?g1) (GPH$object ?g2))
             (and (isomorphism (tau ?g1 ?g2))
                  (2-cell (tau ?g1 ?g2)))))
                  (isomorphic
                      (GPH$opposite (GPH$multiplication ?g1 ?g2))
                      (GPH$multiplication (GPH$opposite ?g2) (GPH$opposite ?g1))))))
```

Coherence

Associative Law

For any three graphs G_0, G_1 and G_2, where G_0 and G_1 are horizontally composable and G_1 and G_2 are horizontally composable – all three graphs share a common class of objects $Obj(G_0) = Obj(G_1) = Obj(G_2) = Obj$ – an associative law for graph multiplication would say that $G_0 \otimes (G_1 \otimes G_2) = (G_0 \otimes G_1) \otimes G_2$. However, this is too strong. What we can say is that the graph $G_0 \otimes (G_1 \otimes G_2)$ and the graph $(G_0 \otimes G_1) \otimes G_2$ are isomorphic. The definition for the appropriate associative graph isomorphic 2-cell

$$\alpha_{G0,\,G1,\,G2} : G_0 \otimes (G_1 \otimes G_2) \to (G_0 \otimes G_1) \otimes G_2$$

is illustrated in Figure 7.

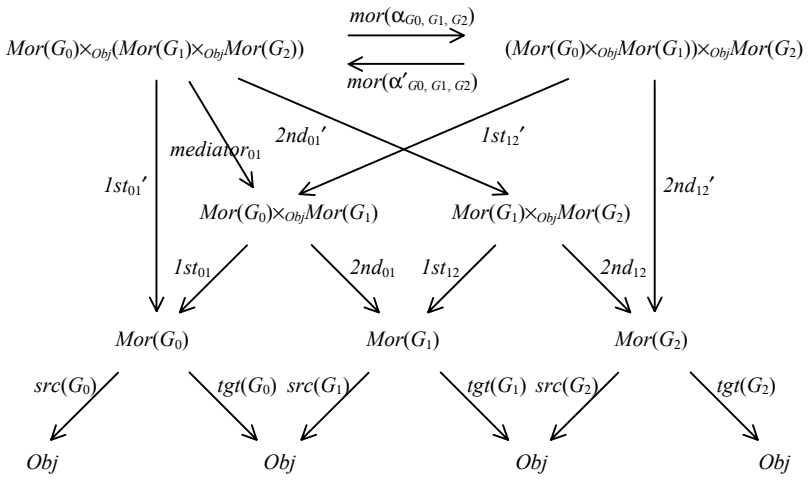


Figure 7: Associativity Graph Isomorphism

In order to define the morphism function

$$mor(\alpha_{G0,\,G1,\,G2}) : Mor(G_0)\times_{Obj}(Mor(G_1)\times_{Obj}Mor(G_2)) \rightarrow (Mor(G_0)\times_{Obj}Mor(G_1))\times_{Obj}Mor(G_2),$$

we need to specify the following auxiliary components for the associative law.

- The cone *first-cone* consists of
  - vertex: $Mor(G_0)\times_{Obj}(Mor(G_1)\times_{Obj}Mor(G_2))$,
  - first function: $1st_{01}' : Mor(G_0)\times_{Obj}(Mor(G_1)\times_{Obj}Mor(G_2)) \rightarrow Mor(G_0)$,
  - second function: $2nd_{01}' \cdot 1st_{12} : Mor(G_0)\times_{Obj}(Mor(G_1)\times_{Obj}Mor(G_2)) \rightarrow Mor(G_1)$, and
  - opspan: opspan of the multiplication $G_0 \otimes G_1$.
- The opspan *opspan12-3* consists of
  - opvertex: *Obj*,
  - opfirst function: $2nd_{01} \cdot tgt(G_1) : Mor(G_0)\times_{Obj}(Mor(G_1) \rightarrow Obj$, and
  - opsecond function: $src(G_2) : Mor(G_2) \rightarrow Obj$.
- The cone *second-cone* consists of
  - vertex: $Mor(G_0)\times_{Obj}(Mor(G_1)\times_{Obj}Mor(G_2))$,
  - first function: $mediator_{01} : Mor(G_0)\times_{Obj}(Mor(G_1)\times_{Obj}Mor(G_2)) \rightarrow Mor(G_0)\times_{Obj}(Mor(G_1)$ of *first-cone*,
  - second function: $2nd_{01}' \cdot 2nd_{12} : Mor(G_0)\times_{Obj}(Mor(G_1)\times_{Obj}Mor(G_2)) \rightarrow Mor(G_2)$, and
  - opspan: *opspan12-3*.

The morphism function $mor(\alpha_{G0,\,G1,\,G2})$ is the mediator function of the pullback cone *second-cone*. For convenience of reference, this morphism is called the *associativity* morphism. This is represented as the ternary CNG function '`(associativity ?g0 ?g1 ?g2)`'.

```
(21) (CNG$function first-cone)
     (CNG$signature first-cone GPH$graph GPH$graph GPH$graph SET.LIM.PBK$cone)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> (exists (?r (SET.LIM.PBK$cone ?r))
                  (= (first-cone ?g0 ?g1 ?g2) ?r))
              (and (= (GPH$object ?g0) (GPH$object ?g1))
                   (= (GPH$object ?g1) (GPH$object ?g2)))))
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (=> (and (= (GPH$object ?g0) (GPH$object ?g1))
                  (= (GPH$object ?g1) (GPH$object ?g2)))
             (and (= (SET.LIM.PBK$vertex (first-cone ?g0 ?g1 ?g2))
                     (SET.LIM.PBK$pullback
                         (GPH$multiplication-opspan ?g0 (GPH$multiplication ?g1 ?g2))))
                  (= (SET.LIM.PBK$first (first-cone ?g0 ?g1 ?g2))
                     (SET.LIM.PBK$projection1
```

```
                    (GPH$multiplication-opspan ?g0 (GPH$multiplication ?g1 ?g2))))
                (= (SET.LIM.PBK$second (first-cone ?g0 ?g1 ?g2))
                   (SET.FTN$composition
                       (SET.LIM.PBK$projection2
                          (GPH$multiplication-opspan ?g0 (GPH$multiplication ?g1 ?g2)))
                       (SET.LIM.PBK$projection1 (GPH$multiplication-opspan ?g1 ?g2))))
                (= (SET.LIM.PBK$cone-diagram (first-cone ?g0 ?g1 ?g2))
                   (GPH$multiplication-opspan ?g0 ?g1)))))

(22) (CNG$function opspan12-3)
     (CNG$signature opspan12-3 GPH$graph GPH$graph GPH$graph SET.LIM.PBK$opspan)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> (exists (?s (SET.LIM.PBK$opspan ?s))
                  (= (opspan12-3 ?g0 ?g1 ?g2) ?s))
              (and (= (GPH$object ?g0) (GPH$object ?g1))
                   (= (GPH$object ?g1) (GPH$object ?g2)))))
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (=> (and (= (GPH$object ?g0) (GPH$object ?g1))
                  (= (GPH$object ?g1) (GPH$object ?g2)))
             (and (= (SET$opvertex (opspan12-3 ?g0 ?g1 ?g2))
                     (GPH$object ?g1))
                  (= (SET$opfirst (opspan12-3 ?g0 ?g1 ?g2))
                     (SET.FTN$composition
                         (SET.LIM.PBK$projection2 (GPH$multiplication-opspan ?g0 ?g1))
                         (target ?g1)))
                  (= (SET$opsecond (opspan12-3 ?g0 ?g1 ?g2))
                     (source ?g2)))))

(23) (CNG$function second-cone)
     (CNG$signature second-cone GPH$graph GPH$graph GPH$graph SET.LIM.PBK$cone)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> (exists (?r (SET.LIM.PBK$cone ?r))
                  (= (second-cone ?g0 ?g1 ?g2) ?r))
              (and (= (GPH$object ?g0) (GPH$object ?g1))
                   (= (GPH$object ?g1) (GPH$object ?g2)))))
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (=> (and (= (GPH$object ?g0) (GPH$object ?g1))
                  (= (GPH$object ?g1) (GPH$object ?g2)))
             (and (= (SET$vertex (second-cone ?g0 ?g1 ?g2))
                     (SET.LIM.PBK$pullback
                         (GPH$multiplication-opspan ?g0 (GPH$multiplication ?g1 ?g2))))
                  (= (SET.LIM.PBK$first (second-cone ?g0 ?g1 ?g2))
                     (SET.LIM.PBK$mediator (first-cone ?g0 ?g1 ?g2)))
                  (= (SET.LIM.PBK$second (second-cone ?g0 ?g1 ?g2))
                     (SET.FTN$composition
                         (SET.LIM.PBK$projection2
                            (GPH$multiplication-opspan ?g0 (GPH$multiplication ?g1 ?g2)))
                         (SET.LIM.PBK$projection2 (GPH$multiplication-opspan ?g1 ?g2))))
                  (= (SET.LIM.PBK$cone-diagram (second-cone ?g0 ?g1 ?g2))
                     (opspan12-3 ?g0 ?g1 ?g2)))))

(24) (CNG$function alpha)
     (CNG$signature alpha GPH$graph GPH$graph GPH$graph graph-morphism)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (<=> (exists (?h (graph-morphism ?h))
                  (= (alpha ?g0 ?g1 ?g2) ?h))
              (and (= (GPH$object ?g0) (GPH$object ?g1))
                   (= (GPH$object ?g1) (GPH$object ?g2)))))
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (=> (and (= (GPH$object ?g0) (GPH$object ?g1))
                  (= (GPH$object ?g1) (GPH$object ?g2)))
             (and (= (source (alpha ?g0 ?g1 ?g2))
                     (GPH$multiplication ?g0 (GPH$multiplication ?g1 ?g2)))
                  (= (target (alpha ?g0 ?g1 ?g2))
                     (GPH$multiplication (GPH$multiplication ?g1 ?g0) ?g2))
                  (= (object (alpha ?g0 ?g1 ?g2))
                     (SET.FTN$identity (GPH$object ?g0)))
                  (= (morphism (alpha ?g0 ?g1 ?g2))
                     (SET.LIM.PBK$mediator (second-cone ?g0 ?g1 ?g2)))))))
```

```
(25) (CNG$function associativity)
     (CNG$signature associativity GPH$graph GPH$graph GPH$graph SET.FTN$function)
     (forall (?g0 (GPH$graph ?g0) ?g1 (GPH$graph ?g1) ?g2 (GPH$graph ?g2))
         (=> (and (= (GPH$object ?g0) (GPH$object ?g1))
                  (= (GPH$object ?g1) (GPH$object ?g2)))
             (= (associativity ?g0 ?g1 ?g2) (morphism (alpha ?g0 ?g1 ?g2)))))
```

The oppositely directed graph morphism $\alpha' : (G_0 \otimes G_1) \otimes G_2 \to G_0 \otimes (G_1 \otimes G_2)$ can be defined in a similar fashion, and, based upon uniqueness of the pullback mediator function, the two can be shown to be inverses. In addition, the associative coherence theorem in Diagram 3 can be proven.

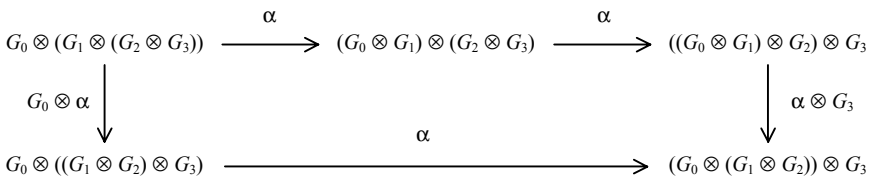


Diagram 3: Associativity Coherence

Unit Laws

For any graph G the unit laws for graph multiplication would say that $1_{Obj(G)} \otimes G = G = G \otimes 1_{Obj(G)}$. However, these are too strong. What we can say is that the graphs $1_{Obj(G)} \otimes G$ and G are isomorphic, and that the graphs $G \otimes 1_{Obj(G)}$ and G are isomorphic. The definitions for the appropriate graph isomorphic 2-cells, *left unit* $\lambda_G : 1_{Obj(G)} \otimes G \to G$ and *right unit* $\rho_G : G \otimes 1_{Obj(G)} \to G$, are illustrated in Figure 8.

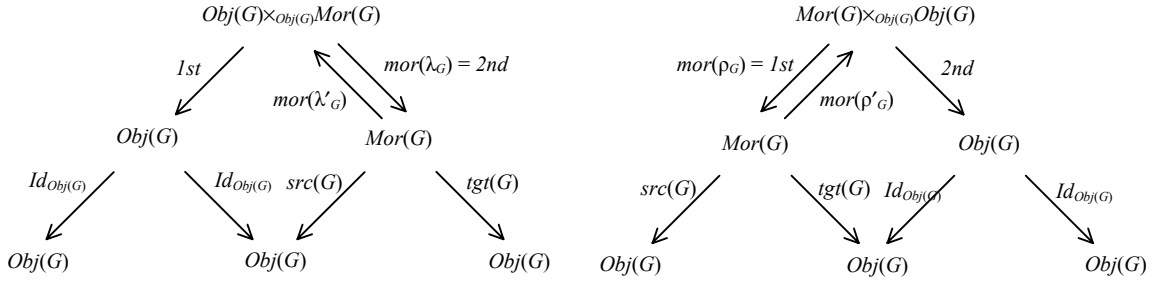


Figure 8: Left Unit and Right Unit Graph Isomorphisms

Here is the KIF formulation for the unit isomorphisms.

```
(26) (CNG$function left)
     (CNG$signature left GPH$graph graph-morphism)
     (forall (?g (GPH$graph ?g))
         (and (= (source (left ?g))
                 (GPH$multiplication (GPH$unit (GPH$object ?g)) ?g))
              (= (target (left ?g)) ?g)
              (= (object (left ?g)) (SET.FTN$identity (GPH$object ?g)))
              (= (morphism (left ?g))
                 (SET.LIM.PBK$projection2
                     (GPH$multiplication-opspan (GPH$unit (GPH$object ?g)) ?g)))))

(27) (CNG$function right)
     (CNG$signature right GPH$graph graph-morphism)
     (forall (?g (GPH$graph ?g))
         (and (= (source (right ?g))
                 (GPH$multiplication ?g (GPH$unit (GPH$object ?g))))
              (= (target (right ?g)) ?g)
              (= (object (right ?g))
                 (SET.FTN$identity (GPH$object ?g)))
              (= (morphism (right ?g))
                 (SET.LIM.PBK$projection1
```

```
(GPH$multiplication-opspan (GPH$unit (GPH$object ?g)) ?g)))))
```

An oppositely directed graph morphism $\lambda' : G \to 1_{Obj(G)} \otimes G$ can be defined, whose morphism function $mor(\lambda') : Mor(G) \to Obj(G) \times_{Obj(G)} Mor(G)$ is the mediator function for a pullback cone over the opspan associated with $1_{Obj(G)} \otimes G$, whose vertex is $Mor(G)$, whose first function is $src(G)$ and whose second function is $Id_{Mor(G)}$. Based upon uniqueness of the pullback mediator function, this can be shown to be inverse to λ. Similarly, an oppositely directed graph morphism $\rho' : G \to G \otimes 1_{Obj(G)}$ can be defined and shown to be the inverse to ρ. In addition, the unit coherence theorem and identity theorem in Diagram 4 can be proven.

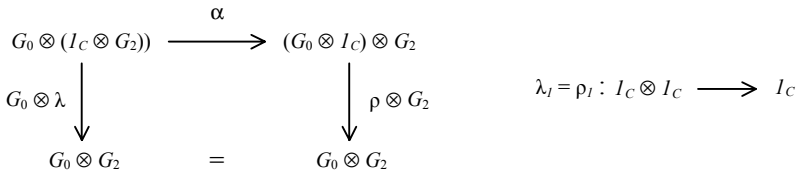


Diagram 4: Unit Coherence

The Namespace of Large Categories

CAT

Basics

- A *category* C can be thought of as a special kind of graph $|C|$ – a graph with monoidal properties. More precisely, a category $C = \langle C, \mu_C, \eta_C \rangle$ is a monoid in the 2-dimensional quasi-category of (large) graphs and graph morphisms. This means that it consists of a graph $|C|$, a *composition* graph morphism $\mu_C : |C| \otimes |C| \to |C|$ and an *identity* graph morphism $\eta_C : I_{obj(C)} \to |C|$, both with the identity object function $id_{obj(C)}$. Table 1 gives the notation for the composition function $\circ^C = mor(\mu_C)$ and the identity function $id^C = mor(\eta_C)$ – these are the morphism functions of the composition and identity graph morphisms.

$\circ^C : mor(C) \times_{obj(C)} mor(C) \to mor(C)$	$id^C : obj(C) \to mor(C)$
$(m_1, m_2) \mapsto m_1 \circ m_2$	$o \mapsto id_o$

Table 1: Elements of Monoidal Structure

Axioms (1–6) give the KIF representation for a category. The unary CNG function '`underlying`' in axiom (2) gives the *underlying* graph of a category. The SET function '`(composition ?c)`' of axiom (4) provides for an associative composition of morphisms in the category – it operates on any two morphisms that are composable, in the sense that the target object of the first is equal to the source object of the second, and returns a well-defined (composition) morphism. The SET function '`(identity ?c)`' in axiom (6) provides identities – it associates a well-defined (identity) morphism with each object in the category. The unary CNG functions '`composition`' and '`identity`' have the category as a parameter. Categories are determined by their (underlying, mu, eta) triples, and hence by their (underlying, composition, identity) triples.

```
(1) (CNG$conglomerate category)

(2) (CNG$function underlying)
    (CNG$signature underlying category GPH$graph)

(3) (CNG$function mu)
    (CNG$signature mu category GPH.MOR$2-cell)
    (forall (?c (category ?c))
        (and (= (GPH.MOR$source (mu ?c))
                (GPH$multiplication (underlying ?c) (underlying ?c)))
             (= (GPH.MOR$target (mu ?c))
                (underlying ?c))))

(4) (CNG$function composition)
    (CNG$signature composition category SET.FTN$function)
    (forall (?c (category ?c))
        (= (composition ?c)
           (GPH.MOR$morphism (mu ?c))))

(5) (CNG$function eta)
    (CNG$signature eta category GPH.MOR$2-cell)
    (forall (?c (category ?c))
        (and (= (GPH.MOR$source (eta ?c))
                (GPH$unit (GPH$object (underlying ?c))))
             (= (GPH.MOR$target (eta ?c))
                (underlying ?c))))

(6) (CNG$function identity)
    (CNG$signature identity category SET$function)
    (forall (?c (category ?c))
        (= (identity ?c)
           (GPH.MOR$morphism (eta ?c))))

    (forall (?c1 (category ?c1) ?c2 (category ?c2))
```

```
        (=> (and (= (underlying ?c1) (underlying ?c2))
                 (= (mu ?c1) (mu ?c2))
                 (= (eta ?c1) (eta ?c2)))
            (= ?c1 ?c2)))
```

- For convenience in the language used for categories, in axioms (7–14) we rename the object and morphism classes, the source and target functions, the class of composable pairs of morphisms, and the first and second functions in the setting of categories.

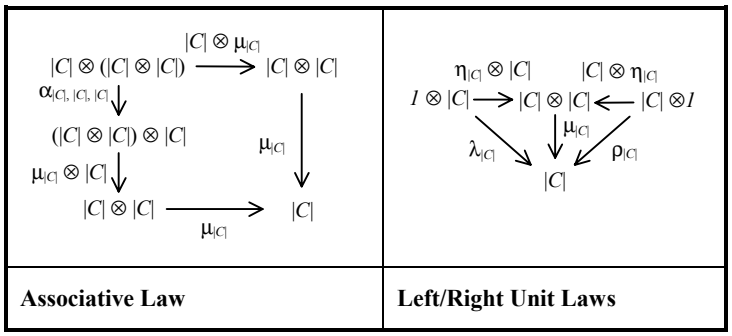
```
(7) (CNG$function object)
    (CNG$signature object category SET$class)
    (forall (?c (category ?c))
        (= (object ?c)
           (GPH$object (underlying ?c))))

(8) (CNG$function morphism)
    (CNG$signature morphism category SET$class)
    (forall (?c (category ?c))
        (= (morphism ?c)
           (GPH$morphism (underlying ?c))))

(9) (CNG$function source)
    (CNG$signature source category SET.FTN$function)
    (forall (?c (category ?c))
        (= (source ?c)
           (GPH$source (underlying ?c))))

(10) (CNG$function target)
     (CNG$signature source category SET.FTN$function)
     (forall (?c (category ?c))
         (= (target ?c)
            (GPH$target (underlying ?c))))

(11) (CNG$function composable-opspan)
     (CNG$signature composable-opspan category SET.LIM.PBK$opspan)
     (forall (?c (category ?c))
         (= (composable-opspan ?c)
            (GPH$multiplication-opspan (underlying ?c) (underlying ?c))))

(12) (CNG$function composable)
     (CNG$signature composable category SET$class)
     (forall (?c (category ?c))
         (= (composable ?c)
            (GPH$morphism
                (GPH$multiplication (underlying ?c) (underlying ?c)))))

(13) (CNG$function first)
     (CNG$signature first category SET.FTN$function)
     (forall (?c (category ?c))
         (= (first ?c)
            (GPH$source
                (GPH$multiplication (underlying ?c) (underlying ?c)))))

(14) (CNG$function second)
     (CNG$signature second category SET$function)
     (forall (?c (category ?c))
         (= (second ?c)
            (GPH$target
                (GPH$multiplication (underlying ?c)(underlying ?c)))))
```

- By the definitions of graph morphisms, graph multiplication and graph units, for any category *C* these operations satisfy the typing constraints listed in Table 2.

$\circ^C \cdot src(C) = 1^{st}(C) \cdot src(C)$
$\circ^C \cdot tgt(C) = 2^{nd}(C) \cdot tgt(C)$
$id^C \cdot src(C) = id_{obj(C)} = id^C \cdot tgt(C)$

Table 2: Preservation of Source and Target

- Table 3 contains commutative diagrams involving the μ and η graph morphisms of categories and the coherence graph morphisms α, λ and ρ. The commutative diagram on the left represents the *associative law* for composition, and the commutative diagrams on the right represent the left and right *unit laws* for identity.



Associative Law	**Left/Right Unit Laws**

Table 3: Laws of Monoidal Structure

- Axiom (15) represents the associative law in KIF. This is an important axiom, since the correct expression of (15) motivated the ontology for graphs and graph morphisms, the representation of categories as monoids in the 2-dimensional category of large graphs, and in particular the coherence axiomatization. Axiom (16) represents the unit laws in KIF. Both are expressed at the level of graph morphisms. Using composition and identity, these could also be expressed at the level of SET functions, as in Table 6.

```
(15) (forall (?c (category ?c))
         (= (GPH.MOR$composition
                (GPH.MOR$multiplication
                    (GPH.MOR$identity (underlying ?c))
                    (mu ?c))
                (mu ?c))
            (GPH.MOR$composition
                (GPH.MOR$composition
                    (GPH.MOR$alpha
                        (underlying ?c) (underlying ?c) (underlying ?c))
                    (GPH.MOR$multiplication
                        (mu ?c)
                        (GPH.MOR$identity (underlying ?c))))
                (mu ?c))))

(16) (forall (?c (category ?c))
         (and (= (GPH.MOR$composition
                     (GPH.MOR$multiplication
                         (eta ?c)
                         (GPH.MOR$identity (underlying ?c)))
                     (mu ?c))
                 (GPH.MOR$left (underlying ?c)))
              (= (GPH.MOR$composition
                     (GPH.MOR$multiplication
                         (GPH.MOR$identity (underlying ?c))
                         (eta ?c))
                     (mu ?c))
                 (GPH.MOR$right (underlying ?c))))))
```

- Table 4 is derivative – it represents the associative and unit laws in terms of the composition and identity functions.

Associative law:	$(m_1 \circ^C m_2) \circ^C m_3 = m_1 \circ^C (m_2 \circ^C m_3)$
Identity laws:	$id^C_a \circ^C m = m = m \circ^C id^C_b$

Table 4: Laws of Monoidal Structure Redux

Additional Categorical Structure

Particular categories may have additional structure. This is true for the categories expressed by the IFF lower metalevel namespaces. Here is the KIF formalization for some of this additional structure.

- A category C is small when its underlying graph is small.

```
(17) (CNG$conglomerate small)
     (CNG$subconglomerate small category)
     (forall (?c (category ?c))
         (<=> (small ?c)
              (GPH$small (underlying ?c))))
```

- To each category C, there is an *opposite category* $C^{op} = \langle C, \mu_C, \eta_C \rangle^{op} = \langle C^{op}, \tau_{C,C} \cdot \mu_C{}^{op}, \eta_C{}^{op} \rangle$. Since all categorical notions have their duals, the opposite category can be used to decrease the size of the axiom set. The objects of C^{op} are the objects of C, and the morphisms of C^{op} are the morphisms of C. However, the source and target of a morphism are reversed: $src(C^{op})(m) = tgt(C)(m)$ and $tgt(C^{op})(m) = src(C)(m)$. The composition is defined by $m_2 \circ^{op} m_1 = m_1 \circ m_2$, and the identity is $id^{op}_o = id_o$. The type restriction axioms in (18) specify the opposite operation on categories.

```
(18) (CNG$function opposite)
     (CNG$signature opposite category category)
     (forall (?c (category ?c))
         (and (= (underlying (opposite ?c))
                 (GPH$opposite (underlying ?c))
              (= (mu (opposite ?c))
                 (GPH.MOR$composition
                     (GPH.MOR$tau (underlying ?c) (underlying ?c))
                     (GPH.MOR$opposite (mu ?c))))
              (= (eta (opposite ?c))
                 (GPH.MOR$opposite (eta ?c))))))
```

- Part of the fact that opposite forms an involution is the theorem that $(C^{op})^{op} = C$.

```
     (forall (?c (category ?c))
         (= (opposite (opposite ?c)) ?c))
```

- It is sometime convenient to have a name for the pair of classes $\langle$`(object ?c)`, `(object ?c)`$\rangle$. This is called '`(object-pair ?c)`'.

```
(19) (CNG$function object-pair)
     (CNG$signature object-pair category SET.LIM.PRD$pair)
     (forall (?c (category ?c))
         (and (= (SET.LIM.PRD$class1 (object-pair ?c)) (object ?c))
              (= (SET.LIM.PRD$class2 (object-pair ?c)) (object ?c))))

(20) (CNG$function object-binary-product)
     (CNG$signature object-binary-product category SET.class)
     (forall (?c (category ?c))
         (= (object-binary-product ?c)
            (SET.LIM.PRD$binary-product (object-pair ?c))))

(21) (CNG$function morphism-pair)
     (CNG$signature morphism-pair category SET.LIM.PRD$pair)
     (forall (?c (category ?c))
         (and (= (SET.LIM.PRD$class1 (morphism-pair ?c)) (morphism ?c))
              (= (SET.LIM.PRD$class2 (morphism-pair ?c)) (morphism ?c))))

(22) (CNG$function morphism-binary-product)
     (CNG$signature morphism-binary-product category SET.class)
     (forall (?c (category ?c))
         (= (morphism-binary-product ?c)
            (SET.LIM.PRD$binary-product (morphism-pair ?c))))
```

- For any two objects o_1 and o_2 in a category C, the *hom-set* $C(o_1, o_2)$ consists of all morphisms with source o_1 and target o_2:

  $C(o_1, o_2) = \{m \mid m \in mor(C),\ src(C)(m) = o_1 \text{ and } tgt(C)(m) = o_2\} \subseteq mor(C)$.

```
(21) (CNG$function source-target)
```

```
        (CNG$signature source-target category SET.FTN$function)
        (forall (?c (category ?c))
            (and (= (SET.FTN$source (source-target ?c)) (morphism ?c))
                 (= (SET.FTN$target (source-target ?c)) (object-binary-product ?c))
                 (= (source-target ?c)
                    ((SET.LIM.PRD$pairing (object-pair ?c)) (source ?c) (target ?c)))))

    (22) (CNG$function object-hom)
        (CNG$signature object-hom category SET.FTN$function)
        (forall (?c (category ?c))
            (and (= (SET.FTN$source (object-hom ?c)) (object-binary-product ?c))
                 (= (SET.FTN$target (object-hom ?c)) (SET$power (morphism ?c)))
                 (= (object-hom ?c)
                    (SET.FTN$fiber (source-target ?c)))))

    (23) (CNG$function morphism-hom)
        (CNG$signature morphism-hom category CNG$function)
        (forall (?c (category ?c))
            (and (CNG$signature (morphism-hom ?c)
                     (morphism-binary-product ?c) SET.FTN$function)
                 (forall (?m1 ((morphism c) ?m1) ?m2 ((morphism c) ?m2))
                     (and (= (SET.FTN$source ((morphism-hom ?c) [?m1 ?m2]))
                             ((object-hom ?c) [((target ?c) ?m1) ((source ?c) ?m2)]))
                          (= (SET.FTN$target ((morphism-hom ?c) [?m1 ?m2]))
                             ((object-hom ?c) [((souce ?c) ?m1) ((target ?c) ?m2)]))
                          (forall (?m (((object-hom ?c)
                                        [((target ?c) ?m1) ((source ?c) ?m2)]) ?m))
                              (= (((morphism-hom ?c) [?m1 ?m2])) ?m)
                                 ((composition ?c)
                                     [((composition ?c) [?m1 ?m]) ?m2])))))))
```

- A *parallel pair* of morphisms in a category C is a pair of morphisms with the same source and target objects. This equivalence relation is the kernel of the source-target function.

```
(24) (CNG$function parallel-pair)
    (CNG$signature parallel-pair category REL.ENDO$equivalence-relation)
    (forall (?c (category ?c))
        (= (parallel-pair ?c) (SET.LIM.EQU$kernel (source-target ?c))))
```

- There are classes of left-composability and right-composability, and functions of left-composition and right-composition. Left-composition by morphism m_1 is the operation: $m_2 \mapsto m_1 \cdot m_2$.

```
(25) (CNG$function left-composable)
    (CNG$signature left-composable category SET.FTN$function)
    (forall (?c (category ?c))
        (and (= (SET.FTN$source (left-composable ?c)) (morphism ?c))
             (= (SET.FTN$target (left-composable ?c)) (SET$power (morphism ?c)))
             (= (left-composable ?c) (SET.LIM.PBK$fiber12 (composable-opspan ?c)))))

(26) (KIF$function left-composition)
    (KIF$signature left-composition category CNG$function)
    (forall (?c (category ?c))
      (and (CNG$signature (left-composition ?c) (morphism ?c) SET.FTN$function)
           (forall (?m1 ((morphism ?c) ?m1))
             (and (= (SET.FTN$source ((left-composition ?c) ?m1))
                     ((left-composable ?c) ?m1))
                  (= (SET.FTN$target ((left-composition ?c) ?m1))
                     (morphism ?c))
                  (= ((left-composition ?c) ?m1)
                     (SET.FTN$composition
                       (SET.FTN$composition
                         ((SET.LIM.PBK$fiber12-embedding (composable-opspan ?c)) ?m1)
                         ((SET.LIM.PBK$fiber-embedding (composable-opspan ?c))
                           ((source ?c) ?m1)))
                       (composition ?c)))))))

(27) (CNG$function right-composable)
    (CNG$signature right-composable category SET.FTN$function)
    (forall (?c (category ?c))
        (and (= (SET.FTN$source (right-composable ?c)) (morphism ?c))
```

```
                (= (SET.FTN$target (right-composable ?c)) (SET$power (morphism ?c)))
                (= (left-composable ?c) (SET.LIM.PBK$fiber21 (composable-opspan ?c)))))

(28) (KIF$function right-composition)
     (KIF$signature right-composition category CNG$function)
     (forall (?c (category ?c))
       (and (CNG$signature (right-composition ?c) (morphism ?c) SET.FTN$function)
            (forall (?m1 ((morphism ?c) ?m1))
              (and (= (SET.FTN$source ((right-composition ?c) ?m1))
                      ((right-composable ?c) ?m1))
                   (= (SET.FTN$target ((right-composition ?c) ?m1))
                      (morphism ?c))
                   (= ((right-composition ?c) ?m1)
                      (SET.FTN$composition
                        (SET.FTN$composition
                          ((SET.LIM.PBK$fiber12-embedding (composable-opspan ?c)) ?m1)
                          ((SET.LIM.PBK$fiber-embedding (composable-opspan ?c))
                            ((source ?c) ?m1)))
                        (composition ?c)))))))
```

- A morphism $m_1 : o_0 \to o_1$ is an *epimorphism* (Axiom 19) in a category C when it is left-cancellable – for any two parallel morphisms $m_2, m_2' : o_1 \to o_2$, the equality $m_1 \circ^C m_2 = m_1 \circ^C m_2'$ implies $m_2 = m_2'$. Equivalently, a morphism $m_1 : o_0 \to o_1$ is an epimorphism (Axiom 19) in a category C when its left composition is an injection. Dually, a morphism $m_2 : o_1 \to o_2$ is a *monomorphism* in a category C when it is right-cancellable – that is, when it is an epimorphism in C^{op}. Axiom (20) uses the duality of the opposite category to express monomorphisms. A morphism is an *isomorphism* (Axiom 21) in a category C when it is both a monomorphism and an epimorphism.

```
(29) (CNG$function epimorphism)
     (CNG$signature epimorphism category SET$class)
     (forall (?c (category ?c))
         (and (SET$subclass (epimorphism ?c) (morphism ?c))
              (forall (?m1 ((morphism ?c) ?m1))
                  (<=> ((epimorphism ?c) ?m1)
                       (SET.FTN$injection ((left-composition ?c) ?m1))))))

(30) (CNG$function monomorphism)
     (CNG$signature monomorphism category SET$class)
     (forall (?c (category ?c))
         (and (SET$subclass (monomorphism ?c) (morphism ?c))
              (forall (?m2 ((morphism ?c) ?m2))
                  (<=> ((monomorphism ?c) ?m1)
                       ((epimorphism (opposite ?c)) ?m1)))))

(31) (CNG$function isomorphism)
     (CNG$signature isomorphism category SET$class)
     (forall (?c (category ?c))
         (and (SET$subclass (monomorphism ?c) (morphism ?c))
              (= (isomorphism ?c)
                 (SET$binary-intersection (epimorphism ?c) (monomorphism ?c))))))
```

- Two objects $o_1, o_2 \in mor(C)$ are *isomorphic* when there is an isomorphism between them; we then use the notation $o_1 \cong_C o_2$.

```
(32) (CNG$function isomorphic)
     (CNG$signature isomorphic category REL.ENDO$endorelation)
     (forall (?c (category ?c))
         (and (REL.ENDO$class (isomorphic ?c)) (object ?c))
              (forall (?o1 ((object ?c) ?o1) ?o2 ((object ?c) ?o2))
                  (<=> ((REL.ENDO$extent (isomorphic ?c)) [?o1 ?o2])
                       (exists (?m ((isomorphism ?c) ?m))
                           (and (= ((source ?c) ?m) ?o1)
                                (= ((target ?c) ?m) ?o2)))))))
```

Examples

Here are some examples of small categories, which can be used as shapes for colimit/limit diagrams.

- The terminal category *set1* has one object 0 and one (identity) morphism 00. A *discrete category* is a category whose morphisms are all identity morphisms. So, essentially a discrete category is just a set (of objects). *set1* is clearly a discrete category.

```
(CAT$category terminal)
(CAT$category unit)
(CAT$category set1)
(= unit terminal)
(= set1 terminal)
(= (CAT$object terminal) SET.LIM$terminal)
(= (CAT$morphism terminal) SET.LIM$terminal)
(= ((CAT$source terminal) set1#00) set1#0)
(= ((CAT$target terminal) set1#00) set1#0)
```

- The discrete category *set2* = • • of two things, is the category, whose set of objects is {0, 1} , whose set of morphisms is {00, 11}, with 00 and 11 being the identity morphisms at objects 0 and 1, respectively. The following KIF represents this category.

```
(CAT$category set2)
((CAT$object set2) set2#0)
((CAT$object set2) set2#1)
((CAT$morphism set2) set2#00)
((CAT$morphism set2) set2#11)
(= ((CAT$identity set2) set2#0) set2#00)
(= ((CAT$identity set2) set2#1) set2#11)
```

- The ordinal category *ord3* (Figure 5) (Mac Lane 1971, p. 11) is the ordinal, whose set of objects is {0, 1, 2}, whose set of morphisms is {00, 11, 22, 01, 12, 02}, with 00, 11 and 22 being the identity morphisms at objects 0, 1 and 2, respectively, and possessing the one nontrivial composition 01 ◦ 12 = 02. The following KIF represents the category *ord3*.

```
(CAT$category ord3)
((CAT$object ord3) ord3#0)
((CAT$object ord3) ord3#1)
((CAT$object ord3) ord3#2)
((CAT$morphism ord3) ord3#00)
((CAT$morphism ord3) ord3#11)
((CAT$morphism ord3) ord3#22)
((CAT$morphism ord3) ord3#01)
((CAT$morphism ord3) ord3#12)
((CAT$morphism ord3) ord3#02)
(= ((CAT$source ord3) ord3#01) ord3#0)
(= ((CAT$target ord3) ord3#01) ord3#1)
(= ((CAT$source ord3) ord3#12) ord3#1)
(= ((CAT$target ord3) ord3#12) ord3#2)
(= ((CAT$source ord3) ord3#02) ord3#0)
(= ((CAT$target ord3) ord3#02) ord3#2)
(= ((CAT$identity ord3) ord3#0) ord3#00)
(= ((CAT$identity ord3) ord3#1) ord3#11)
(= ((CAT$identity ord3) ord3#2) ord3#22)
(= ((CAT$composition ord3) ord3#01 ord3#12) ord3#02)
```

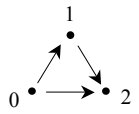


Figure 5: ordinal 3

- The shape category *parpair* = • ⇉ • consists of a parallel pair of edges, whose set of objects is {0, 1}, whose set of morphisms is {00, 11, a_0, a_1}, with 00 and 11 being the identity morphisms at objects 0 and 1, respectively. The following KIF represents this category.

```
(CAT$category parpair)
((CAT$object parpair) parpair#0)
((CAT$object parpair) parpair#1)
((CAT$morphism parpair) parpair#00)
((CAT$morphism parpair) parpair#11)
((CAT$morphism parpair) parpair#a0)
((CAT$morphism parpair) parpair#a1)
(= ((CAT$source parpair) parpair#a0) parpair#0)
(= ((CAT$target parpair) parpair#a0) parpair#1)
```

```
(= ((CAT$source parpair) parpair#a1) parpair#0)
(= ((CAT$target parpair) parpair#a1) parpair#1)
(= ((CAT$identity parpair) parpair#0) parpair#00)
(= ((CAT$identity parpair) parpair#1) parpair#11)
```

- The shape category *span* = • ← • → • consists of a pair of morphisms a_1 and a_2 with common source object 0 and target objects 1 and 2, respectively. The class of objects is $obj(J) = \{0, 1, 2\}$, and the class of morphisms is $mor(J) = \{00, 11, 22, a_1, a_2\}$, with 00, 11 and 22 being the identity morphisms at objects 0, 1 and 2 respectively. Here is the KIF representation of the *span* shape category.

```
(CAT$category span)
((CAT$object span) span#0)
((CAT$object span) span#1)
((CAT$object span) span#2)
((CAT$morphism span) span#00)
((CAT$morphism span) span#11)
((CAT$morphism span) span#22)
((CAT$morphism span) span#a1)
((CAT$morphism span) span#a2)
(= ((CAT$source span) span#a1) span#0)
(= ((CAT$target span) span#a1) span#1)
(= ((CAT$source span) span#a2) span#0)
(= ((CAT$target span) span#a2) span#2)
(= ((CAT$identity span) span#0) span#00)
(= ((CAT$identity span) span#1) span#11)
(= ((CAT$identity span) span#2) span#22)
```

Here are examples of categories defined elsewhere, but asserted to be categories here.

- Two important categories are implicitly defined within the classification namespace in the Model Ontology – Classification the category of classifications and infomorphisms, and Set the category of small sets and their functions. The assertion that "Classification and Set are categories" could not be made in the Model Theory Ontology (the lower metalevel of the IFF Foundation Ontology), since the appropriate functorial machinery is not present there. The Category Theory Ontology provides that machinery. To make that assertion requires that we also describe or identify the components of a category: the underlying graph (object and morphism sets, and source and target functions), and the composition and identity functions (or the mu and eta graph 2-cells). Here we make these assertions in an external namespace. Proofs of some categorical properties, such as associativity of composition, will involve getting further into the details of the specific category, in this case Classification; in particular, the associativity of '`set.ftn$composition`', etc.

```
(CAT$category Classification)
(= (CAT$underlying Classification)   cls.info$classification-graph)
    (= (CAT$object Classification)   cls$classification)
    (= (CAT$morphism Classification) cls.info$infomorphism)
    (= (CAT$source Classification)   cls.info$source)
    (= (CAT$target Classification)   cls.info$target)
(= (CAT$composable Classification)   cls.info$composable)
(= (CAT$first Classification)        cls.info$first)
(= (CAT$second Classification)       cls.info$second)
(= (CAT$composition Classification)  cls.info$composition)
(= (CAT$identity Classification)     cls.info$identity)

(CAT$category Set)
(= (CAT$underlying Set)   set.ftn$set-graph)
    (= (CAT$object Set)   set$set)
    (= (CAT$morphism Set) set.ftn$function)
    (= (CAT$source Set)   set.ftn$source)
    (= (CAT$target Set)   set.ftn$target)
(= (CAT$composable Set)   set.ftn$composable)
(= (CAT$first Set)        set.ftn$first)
(= (CAT$second Set)       set.ftn$second)
(= (CAT$composition Set)  set.ftn$composition)
(= (CAT$identity Set)     set.ftn$identity)
```

The Namespace of Large Functors

FUNC

Basics

- A *functor* $F : C_0 \rightarrow C_1$ from category C_0 to category C_1 (Figure 1) is a morphism of categories. A functor is a special kind of graph morphism $|F| : |C_0| \rightarrow |C_1|$ – a graph morphism that preserves the monoidal properties of the categories. The underlying operator preserves source and target. These functions must preserve source and target in the sense that the diagram in Figure 1 is commutative. However, this follows from properties of graph morphisms. A functor is determined by its associated triple (source, target, underlying).

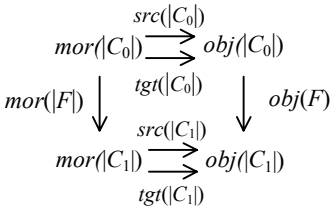


Figure 1: Functor

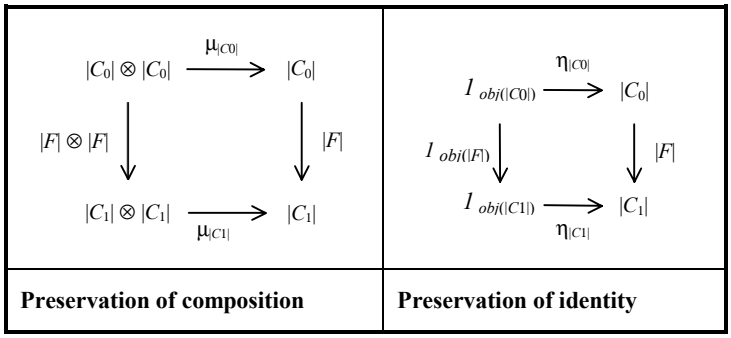
  Axioms (1–4) give the KIF representation for a functor. The unary CNG function '`underlying`' in axiom (4) gives the *underlying* graph morphism of a functor. Functors are determined by their underlying graph morphism.

```
(1) (CNG$conglomerate functor)

(2) (CNG$function source)
    (CNG$signature source functor CAT$category)

(3) (CNG$function target)
    (CNG$signature target functor CAT$category)

(4) (CNG$function underlying)
    (CNG$signature underlying functor GPH.MOR$graph-morphism)

    (forall (?f (functor ?f))
        (and (= (CAT$underlying (source ?f))
                (GPH.MOR$source (underlying ?f)))
             (= (CAT$underlying (target ?f))
                (GPH.MOR$target (underlying ?f)))))

    (forall (?f1 (functor ?f1) ?f2 (functor ?f2))
        (=> (and (= (source ?f1) (source ?f2))
                 (= (target ?f1) (target ?f2))
                 (= (underlying ?f1) (underlying ?f2)))
            (= ?f1 ?f2)))
```

- For convenience in the language used for functors, in axioms (5–6) we rename the object and morphism functions in the setting of functors.

```
(5) (CNG$function object)
    (CNG$signature object functor SET.FTN$function)
    (forall (?f (functor ?f))
        (= (object ?f)
           (GPH.MOR$object (underlying ?f))))

(6) (CNG$function morphism)
    (CNG$signature morphism function SET.FTN$function)
    (forall (?f (functor ?f))
        (= (morphism ?f)
           (GPH.MOR$morphism (underlying ?f))))
```

Table 1: Preservation of monoidal structure



Preservation of composition	**Preservation of identity**

- A functor must preserve monoidal properties – it must preserve identities and compositions in the sense of the commutative diagrams in Table 1. These are commutative diagrams of graph morphisms. The commutative diagram on the left represents preservation of composition, and the commutative diagram on the right represents preservation of identity.
- Axiom (7) represents the preservation of composition law in KIF. Axiom (8) represents preservation of identity in KIF. Both are expressed at the level of graph morphisms.

```
(7) (forall (?f (functor ?f))
        (= (GPH.MOR$composition (mu (source ?f)) (underlying ?f))
           (GPH.MOR$composition
               (GPH.MOR$multiplication (underlying ?f) (underlying ?f))
               (mu (target ?f)))))

(8) (forall (?f (functor ?f))
        (= (GPH.MOR$composition (eta (source ?f)) (underlying ?f))
           (GPH.MOR$composition (unit (object ?f)) (eta (target ?f)))))
```

- Using composition and identity, the associative and unit laws could also be expressed at the level of SET functions, as in Table 2, which is derivative – it represents the preservation of monoidal structure in terms of the composition and identity functions.

Associative law:	$mor(\|F\|)(m_1 \circ^0 m_2) = mor(\|F\|)(m_1) \circ^1 mor(\|F\|)(m_2)$ for all composable pairs of morphisms $m_1, m_2 \in Mor(C_0)$
Identity laws:	$mor(\|F\|)(id^0_o) = id^1_{obj(\|F\|)\,(o)}$ for all objects $o \in Obj(C_0)$

Table 2: Laws of Monoidal Structure Redux

```
(forall (?f (functor ?f))
    (= (SET.FTN$composition (CAT$composition (source ?f)) (morphism ?f))
       (SET.FTN$composition
           ((SET.LIM.PBK$pairing (CAT$composable-opspan (target ?f)))
               (SET.FTN$composition (CAT$first (source ?f)) (morphism ?f))
               (SET.FTN$composition (CAT$second (source ?f)) (morphism ?f)))
           (CAT$composition (target ?f)))))

(forall (?f (functor ?f))
    (= (SET.FTN$composition (CAT$identity (source ?f)) (morphism ?f))
       (SET.FTN$composition (object ?f) (CAT$identity (target ?f)))))
```

Additional Functorial Structure

- Given any category C, there is a *unique functor* $!_C : C \to 1$ to the terminal category – the object and morphism functions are the unique SET functions to the terminal class.

```
(9) (CNG$function unique)
    (CNG$signature unique CAT$category functor)
    (forall (?c (CAT$category ?c))
        (and (= (source (unique ?c)) ?c)
             (= (target (unique ?c)) (CAT$category terminal))
             (= (object (unique ?c)) (SET.FTN$unique (CAT$object ?c)))
             (= (morphism (unique ?c)) (SET.FTN$unique (CAT$morphism ?c)))))
```

- For each category C and each object $o \in C$ there is an *element functor* $elmt_C(o) : 1 \to C$.

```
(10) (CNG$function element)
     (CNG$signature element CAT$category CNG$function)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (element ?c) (CAT$object ?c) functor))
              (forall (?o ((CAT$object ?c) ?o))
                  (and (= (source ((element ?c) ?o)) CAT$terminal)
                       (= (target ((element ?c) ?o)) ?c)
                       (= ((object ((element ?c) ?o)) 0) ?o)
                       (= ((morphism ((element ?c) ?o)) 0) ((CAT$identity ?c) ?o))))))
```

- A category A is said to be a *subcategory* of category B when there is a functor $incl_{A,B} : A \to B$ whose object and morphism functions are injections. Clearly, this provides an partial order on categories.

```
(11) (CNG$relation subcategory)
     (CNG$signature subcategory CAT$category CAT$category)
     (forall (?a (CAT$category ?a) ?b (CAT$category ?b))
         (<=> (subcategory ?a ?b)
              (exists (?f (functor ?f))
                  (and ((source ?f) ?a)
                       ((target ?f) ?b)
                       (SET.FTN$injection (object ?f))
                       (SET.FTN$injection (morphism ?f))))))
```

- To each functor $F : C \to C'$, there is an *opposite functor* $F^{op} : C^{op} \Rightarrow C'^{op}$. The underlying graph morphism of F^{op} is the opposite $|F^{op}| = |F|^{op} : |C_0|^{op} \to |C_1|^{op}$: the object function of F^{op} is the object function of F, and the morphism function of F^{op} is the morphism function of H. However, the source and target categories are the opposite: $src(F^{op}) = src(F)^{op}$ and $tgt(F^{op}) = tgt(F)^{op}$. Axiom (9) specifies an opposite functor.

```
(12) (CNG$function opposite)
     (CNG$signature opposite functor functor)
     (forall (?f (functor ?f))
         (and (= (source (opposite ?f)) (CAT$opposite (source ?f)))
              (= (target (opposite ?f)) (CAT$opposite (target ?f)))
              (= (underlying (opposite ?f)) (GPH.MOR$opposite (object ?h)))
              (= (object (opposite ?f)) (object ?f))
              (= (morphism (opposite ?h)) (morphism ?h))))
```

  An immediate theorem is that the opposite of the opposite of a functor is the original functor.

```
     (forall (?f (functor ?f))
         (= (opposite (opposite ?f)) ?f))
```

- An *opspan* of functors consists of two functors $F : A \to C$ and $G : B \to C$ with a common target category C. Each opspan of functors determines a *comma category* $(F \downarrow G)$, whose objects are triples $\langle a, m, b \rangle$ with $a \in obj(A)$, $b \in obj(B)$ and $m : F(a) \to G(b)$, and whose morphisms

  $\langle u, v \rangle : \langle a_1, m_1, b_1 \rangle \to \langle a_2, m_2, b_2 \rangle$

  are pairs of morphisms $u : a_1 \to a_2$ and $v : b_1 \to b_2$ from the source categories that satisfy the commutative Diagram 1.

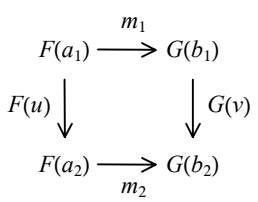


Diagram 1: comma category

Here is the KIF formalization of comma categories.

```
(13) (CNG$conglomerate opspan)

(14) (CNG$function category1)
     (CNG$signature category1 diagram CAT$category

(15) (CNG$function category2)
     (CNG$signature category2 diagram CAT$category)

(16) (CNG$function opvertex)
     (CNG$signature opvertex diagram CAT$category)

(17) (CNG$function opfirst)
     (CNG$signature opfirst diagram functor)

(18) (CNG$function opsecond)
     (CNG$signature opsecond diagram functor)

    (forall (?s (opspan ?s))
       (and (= (SET.FTN$source (opfirst ?s)) (category1 ?s))
            (= (SET.FTN$source (opsecond ?s)) (category2 ?s))
            (= (SET.FTN$target (opfirst ?s)) (opvertex ?s))
            (= (SET.FTN$target (opsecond ?s)) (opvertex ?s))))

    (forall (?s (opspan ?s) ?t (opspan ?t))
       (=> (and (= (opfirst ?s) (opfirst ?t))
                (= (opsecond ?s) (opsecond ?t)))
           (= ?s ?t)))

(19) (CNG$function comma-category)
     (CNG$signature comma-category opspan CAT$category)
     (forall (?s (opspan ?s))
         (and (forall (?o)
                  (<=> ((CAT$object (comma-category ?s)) ?o)
                       (and (KIF$triple ?o)
                            ((CAT$object (category1 ?s)) (?o 1))
                            ((CAT$morphism (opvertex ?s)) (?o 2))
                            ((CAT$object (category2 ?s)) (?o 3))
                            (= ((CAT$source (opvertex ?s)) (?o 2))
                               ((object (opfirst ?s)) (?o 1)))
                            (= ((CAT$target (opvertex ?s)) (?o 2))
                               ((object (opsecond ?s)) (?o 3))))))
              (forall (?m)
                  (<=> ((CAT$morphism (comma-category ?s)) ?m))
                       (and (KIF$pair ?m)
                            ((CAT$morphism (category1 ?s)) (?m 1))
                            (= ((CAT$source (category1 ?s)) (?m 1))
                               ((CAT$source ?m) 1))
                            (= ((CAT$target (category1 ?s)) (?m 1))
                               ((CAT$target ?m) 1))
                            ((CAT$morphism (category2 ?s)) (?m 2))
                            (= ((CAT$source (category2 ?s)) (?m 2))
                               ((CAT$source ?m) 3))
                            (= ((CAT$target (category2 ?s)) (?m 2))
                               ((CAT$target ?m) 3))
                            (= (((CAT$composition (opvertex ?s))
                                    [((CAT$source ?m) 2) (?m 2)])
                               (((CAT$composition (opvertex ?s))
                                    [(?m 1) ((CAT$target ?m) 2)])))))))

(20) (CNG$function objects-under-opspan)
     (CNG$signature objects-under-opspan functor CNG$function)
     (forall (?f (functor ?f))
         (and (CNG$signature (objects-under-opspan ?f)
                  (CAT$object (target ?f)) opspan)
              (forall (?a ((CAT$object (target ?f)) ?a))
                  (and (= (category1 ((objects-under-opspan ?f) ?a)) CAT$terminal)
                       (= (category2 ((objects-under-opspan ?f) ?a)) (source ?f))
                       (= (opvertex ((objects-under-opspan ?f) ?a)) (target ?f))
                       (= (opfirst ((objects-under-opspan ?f) ?a))
```

```
                    ((element (target ?f)) ?a))
                (= (opsecond ((objects-under-opspan ?f) ?a)) ?f)))))
```

```
(21) (CNG$function objects-under)
     (CNG$signature objects-under functor CNG$function)
     (forall (?f (functor ?f))
         (and (CNG$signature (objects-under ?f)
                  (CAT$object (target ?f)) CAT.category)
              (forall (?a ((CAT$object (target ?f)) ?a))
                  (= ((objects-under ?f) ?a)
                     (comma-category ((objects-under-opspan ?f) ?a))))))
```

- For any functor $U : B \to A$ and any object $a \in obj(A)$, a *universal morphism* from a to U (Diagram 2) is a pair $\langle m_a, \tilde{a} \rangle$ consisting of an object $\tilde{a} \in obj(B)$ and a morphism $m_a : a \to U(\tilde{a})$, such that for every pair $\langle m, b \rangle$ consisting of an object $b \in obj(B)$ and a morphism $m : a \to U(b)$, there is a unique morphism $m' : \tilde{a} \to b$ with $m_a \cdot_A U(m') = m$. Equivalently, a universal morphism is an initial object in the comma category $(a \downarrow U)$.

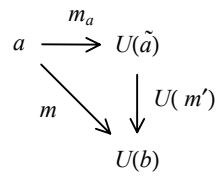


Diagram 2: universal morphism

  Here is the KIF formalism for universal morphisms. For an arbitrary pair $\langle U, a \rangle$ the universal morphism may not exist – in the code below the term '`COL$initial`' refers to a possibly empty class of objects.

```
(22) (CNG$function universal-morphism)
     (CNG$signature universal-morphism functor CNG$function)
     (forall (?f (functor ?f))
         (and (CNG$signature (universal-morphism ?f)
                  (CAT$object (target ?f)) SET.class)
              (forall (?a ((CAT$object (target ?f)) ?a))
                  (= ((universal-morphism ?f) ?a)
                     (COL$initial ((objects-under ?f) ?a))))))
```

Quasi-Category Structure

- A pair of functors F and G is composable when the target category of F is the source category of G. For any composable pair of functors $F : C_0 \to C_1$ and $G : C_1 \to C_2$ there is a *composition functor* $F \cdot G : C_0 \to C_2$. It is defined in terms of the underlying graph morphisms – object and morphism functions are the composition functions of the object and morphism functions of the component functors, respectively. For any category C there is an *identity functor* $Id_C : C \to C$ on that category. Its underlying graph morphisms is the identity on the underlying graph – object and morphism functions are the identity functions on the object and morphism sets of that category, respectively.

  Here is the KIF that formalizes the definitions of composition and identity.

```
(23) (CNG$function composition)
     (CNG$signature composition functor functor functor)
     (forall (?f1 (functor ?f1) ?f2 (functor ?f2))
         (<=> (exists (?f (functor ?f))
                  (= ?f (composition ?f1 ?f2)))
              (= (target ?f1) (source ?f2))))

(24) (forall (?f1 (functor ?f1) ?f2 (functor ?f2))
         (=> (= (target ?f1) (source ?f2))
             (and (= (source (composition ?f1 ?f2))
                     (source ?f1))
                  (= (target (composition ?f1 ?f2))
                     (target ?f2))
                  (= (underlying (composition ?f1 ?f2))
                     (GPH.MOR$composition (underlying ?f1) (underlying ?f2))))))

(25) (CNG$function identity)
     (CNG$signature identity CAT$category functor)

(26) (forall (?c (CAT$category ?c))
         (and (= (source (identity ?c)) ?c)
              (= (target (identity ?c)) ?c)
              (= (underlying (identity ?c))
```

```
                (GPH.MOR$identity (underlying ?c)))))
```

○ Given any category C (to be used as a base category in the colimit namespace) and any category J (to be used as a shape category in the colimit namespace), the *diagonal functor* $\Delta_{J, C} : C \to C^J$ maps an object $o \in obj(C)$ to an associated constant functor $\Delta_{J, C}(o) : J \to C$, which is defined as the functor composition $\Delta_{J, C}(o) = !_J \cdot elmt_C(o)$ – it maps each object $j \in obj(J)$ to the object $o \in obj(C)$ and maps each morphism $n \in mor(J)$ to the identity morphism at o.

```
(27) (KIF$function diagonal)
     (KIF$signature diagonal CAT$category CAT$category CNG$function)
     (forall (?j (CAT$category ?j) ?c (CAT$category ?c))
         (and (CNG$signature (diagonal ?j ?c) (CAT$object ?c) (diagram ?c))
              (forall (?o ((CAT$object ?c) ?o))
                  (and (= ((shape ?c) ((diagonal ?j ?c) ?o)) ?j)
                       (= ((diagonal ?j ?c) ?o)
                          (composition (unique ?j) ((element ?c) ?o)))))))
```

Functor Theorems

○ It can be shown that functor composition satisfies the following associative law

$$(F_1 \circ F_2) \circ F_3 = F_1 \circ (F_2 \circ F_3)$$

for all composable pairs of functors (F_1, F_2) and (F_2, F_3), and that graph morphism identity satisfies the following identity laws

$$Id_{C0} \cdot F = F \text{ and } F = F \cdot Id_{C1}$$

for any functor F: $C_0 \to C_1$ with source category C_0 and target category G_1. Categories as objects and functors as morphisms form a quasi-category ("quasi" since this is at the level of conglomerates in foundations). This has the following expression in an external namespace.

```
(forall (?f1 (FUNC$functor ?f1)
         ?f2 (FUNC$functor ?f2)
         ?f3 (FUNC$functor ?f3))
    (=> (and (= (FUNC$target ?f1) (FUNC$source ?f2))
             (= (FUNC$target ?f2) (FUNC$source ?f3)))
        (= (FUNC$composition (FUNC$composition ?f1 ?f2) ?f3)
           (FUNC$composition ?f1 (FUNC$composition ?f2 ?f3)))))

(forall (?f (FUNC$functor ?f))
    (and (= (FUNC$composition (FUNC$identity (FUNC$source ?f)) ?f) ?f)
         (= (FUNC$composition ?f (FUNC$identity (FUNC$target ?f))) ?f)))
```

Examples

○ For any category C and any there is a counique functor from the initial category to

Here are examples of functors defined elsewhere, but asserted to be functors here.

o Three important functors are implicitly defined within the classification namespace in the Model Ontology. The SET functions '`cls$instance`' and '`cls.info$instance`' represent the object and morphism components of the *instance functor* *inst* : Classification $\to$ Set^{op} (the opposite of the category Set) – the object function takes a classification to its instance set and the morphism function takes an infomorphism to its instance function. Dually, the SET functions '`cls$type`' and '`cls.info$type`' represent the object and morphism components of the *type functor* *typ* : Classification $\to$ Set – the object function takes a classification to its type set and the morphism function takes an infomorphism to its type function. The SET functions '`cls$instance-power`' and '`cls.info$instance-power`' represent the object and morphism components of the contravariant *instance power functor* *pow* : Set^{op} $\to$ Classification. The assertion that "*inst*, *typ* and *pow* are functors" could not be made in the Model Theory Ontology (the lower metalevel of the IFF Foundation Ontology), since the appropriate functorial machinery is not present there. The Category Theory Ontology provides that machinery. To make that assertion requires that we also describe or identify the components of a functor: the source and target categories, and the underlying graph morphism (object and morphism functions). Here we make these assertions in an external namespace. Proofs of some functorial properties, such as

preservation of associativity, will involve getting further into the details of the specific category, in this case Classification; in particular, the instance and type function components of an infomorphism, and the associativity of '`set.ftn$composition`'.

```
(FUNC$functor inst)
(= (FUNC$source inst)     Classification)
(= (FUNC$target inst)     (opposite Set))
(= (FUNC$underlying inst) cls.info$instance-graph-morphism)
    (= (FUNC$object inst)   cls$instance)
    (= (FUNC$morphism inst) cls.info$instance)

(FUNC$functor typ)
(= (FUNC$source typ)     Classification)
(= (FUNC$target typ)     Set)
(= (FUNC$underlying typ) cls.info$type-graph-morphism)
    (= (FUNC$object typ)   cls$type)
    (= (FUNC$morphism typ) cls.info$type)

(FUNC$functor instance-power)
(= (FUNC$source instance-power) (CAT$opposite set))
(= (FUNC$target instance-power) classification)
(= (FUNC$underlying instance-power) cls.info$instance-power-graph-morphism)
    (= (FUNC$object instance-power) cls$instance-power)
    (= (FUNC$morphism instance-power) cls.info$instance-power)
```

The Namespace of Large Natural Transformations

NAT

Natural Transformations

- Suppose that two functors $F_0, F_1 : C_0 \to C_1$ share a common source category C_0 and a common target category C_1. A natural transformation τ from source functor F_0 to target functor F_1, written 1-dimensionally as $\tau : F_0 \Rightarrow F_1 : C_0 \to C_1$ or visualized 2-dimensionally in Figure 1, is a collection of morphisms in the target category parameterized by objects in the source category that link the functorial images:

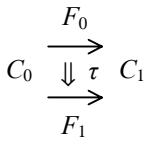


Figure 1: Natural Transformation

$\tau = \{\tau_o : F_0(o) \to F_1(o) \mid o \in Obj(C_0)\}$.

A natural transformation is determined by its (source-functor, target-functor, component) triple. The KIF encoding for the declaration of a natural transformation is as follows.

```
(1) (CNG$conglomerate natural-transformation)

(2) (CNG$function source-functor)
    (CNG$signature source-functor natural-transformation FUNC$functor)

(3) (CNG$function target-functor)
    (CNG$signature target-functor natural-transformation FUNC$functor)

(4) (CNG$function source-category)
    (CNG$signature source-category natural-transformation CAT$category)

(5) (CNG$function target-category)
    (CNG$signature target-category natural-transformation CAT$category)

    (forall (?tau (natural-transformation ?tau))
        (and (= (FUNC$source (source-functor ?tau))
                (source-category ?tau))
                (FUNC$source (target-functor ?tau))
                (source-category ?tau))
             (= (FUNC$target (source-functor ?tau))
                (target-category ?tau))
                (FUNC$target (target-functor ?tau))
                (target-category ?tau))))

(6) (CNG$function component)
    (CNG$signature component natural-transformation SET.FTN$function)

    (forall (?tau ?o ?m)
        (and (= (SET.FTN$source (component ?tau))
                (CAT$object (source-category ?tau)))
             (= (SET.FTN$target (component ?tau))
                (CAT$morphism (target-category ?tau)))))

    (forall (?n1 (natural-transformation ?n1) ?n2 (natural-transformation ?n2))
        (=> (and (= (source-functor ?n1) (source-functor ?n2))
                 (= (target-functor ?n1) (target-functor ?n2))
                 (= (component ?n1) (component ?n2)))
            (= ?n1 ?n2)))
```

The source and target objects of the components of a natural transformation are image objects of the source and target functors:

$src(C_1)(\tau(o)) = obj(F_0)(o)$ and $tgt(C_1)(\tau(o)) = obj(F_1)(o)$

for any object $o \in Obj(C_0)$.

Here is the KIF representation for component source and target.

```
    (forall (?tau (natural-transformation ?tau))
        (and (= (SET.FTN$composition
```

```
            (component ?tau)
            (CAT$source (target-category ?tau)))
        (FUNC$object (source-functor ?tau)))
    (= (SET.FTN$composition
            (component ?tau)
            (CAT$target (target-category ?tau)))
        (FUNC$object (target-functor ?tau)))))
```

The components of a natural transformation interact with the functorial images by satisfying the commutative Diagram 1:

$$mor(F_0)(m) \circ^1 \tau(\partial_1^0(m)) = \tau(\partial_0^0(m)) \circ^1 mor(F_1)(m)$$

for any morphism $m \in Mor(C_0)$. Here is the (somewhat complicated) KIF representation for this interaction, which uses pullback pairing with respect to the composition opspan of the target category of the natural transformation. Again, this is the logical KIF formalization of the commutative diagram for a natural transformation (Diagram 1).

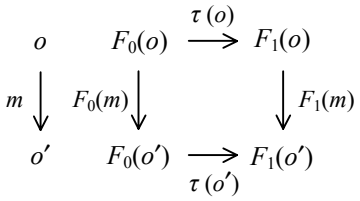


Diagram 1: Natural Transformation

```
(forall (?tau (natural-transformation ?tau))
    (= (SET.FTN$composition
            ((SET.LIM.PBK$pairing (CAT$composable-opspan (target-category ?tau)))
                (SET.FTN$composition
                    (CAT$source (source-category ?tau))
                    (component ?tau)))
                (FUNC$morphism (target-functor ?tau))
            (CAT$composition (target-category ?tau)))
        (SET.FTN$composition
            ((SET.LIM.PBK$pairing (CAT$composable-opspan (target-category ?tau)))
                (FUNC$morphism (source-functor ?tau))
                (SET.FTN$composition
                    (CAT$target (source-category ?tau))
                    (component ?tau)))
            (CAT$composition (target-category ?tau)))))
```

2-Dimensional Category Structure

- A pair of natural transformations σ and τ is *vertically composable* when the target functor of the first is the source functor of the second, $\sigma : F_0 \Rightarrow F_1 : C_0 \to C_1$ and $\tau : F_1 \Rightarrow F_2 : C_0 \to C_1$. The *vertical composition* $\sigma \bullet \tau : F_0 \Rightarrow F_2 : C_0 \to C_1$ of two vertically composable natural transformations is defined as morphism composition in the target category: $\sigma \bullet \tau\ (o) = \sigma(o) \cdot \tau(o)$ for any object $o \in Obj(C_0)$. Diagram 2 illustrates vertical composition.

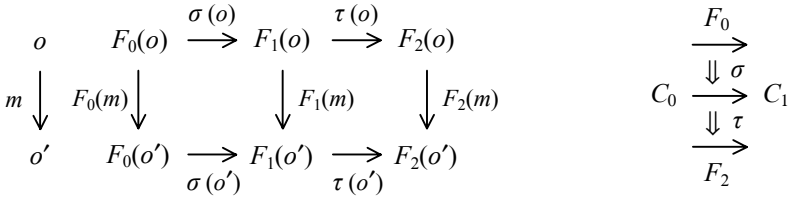


Diagram 2: Vertical composition

- For any functor $F : C_0 \to C_1$ there is a *vertical identity* natural transformation $1_F : F \Rightarrow F : C_0 \to C_1$ defined in terms of identity morphisms in the target category: $1_F\ (o) = id_{obj(F)(o)}$ for any object $o \in Obj(C_0)$. Here is the KIF representation for vertical composition.

```
(7) (CNG$function vertical-composition)
    (CNG$signature vertical-composition
        natural-transformation natural-transformation natural-transformation)

    (forall (?sigma (natural-transformation ?sigma)
             ?tau (natural-transformation ?tau))
        (<=> (exists (?n (natural-transformation ?n))
                 (= ?n (vertical-composition ?sigma ?tau))))
```

```
            (= (target-functor ?sigma) (source-functor ?tau))))

    (forall (?sigma (natural-transformation ?sigma)
             ?tau (natural-transformation ?tau))
        (=> (= (target-functor ?sigma) (source-functor ?tau))
            (and (= (source-functor (vertical-composition ?sigma ?tau))
                    (source-functor ?sigma))
                 (= (target-functor (vertical-composition ?sigma ?tau))
                    (target-functor ?tau))
                 (= (source-category (vertical-composition ?sigma ?tau))
                    (source-category ?sigma))
                 (= (target-category (vertical-composition ?sigma ?tau))
                    (target-category ?sigma)))))

    (forall (?sigma (natural-transformation ?sigma)
             ?tau (natural-transformation ?tau))
        (=> (= (target-functor ?sigma) (source-functor ?tau))
            (= (component (vertical-composition ?sigma ?tau))
               (SET.FTN$composition
                   ((SET.LIM.PBK$pairing
                        (CAT$composable-opspan (target-category ?sigma)))
                    (component ?sigma))
                    (component ?tau)))
                (CAT$composition (target-category ?sigma))))))

(8) (CNG$function vertical-identity)
    (CNG$signature vertical-identity FUNC$functor natural-transformation)

    (forall (?f (FUNC$functor ?f))
        (and (= (source-functor (vertical-identity ?f)) ?f)
             (= (target-functor (vertical-identity ?f)) ?f)))

    (forall (?f (FUNC$functor ?f) ?o ((CAT$object (FUNC$source ?f)) ?o))
        (= (component (vertical-identity ?f))
           (SET.FTN$composition (FUNC$object ?f) (CAT$identity (FUNC$target ?f)))))
```

- A pair of natural transformations σ and τ is *horizontally composable* when the target category of the first is the source category of the second, $\sigma : F_0 \Rightarrow F_1 : C_0 \to C_1$ and $\tau : G_0 \Rightarrow G_1 : C_1 \to C_2$. The *horizontal composition* $\sigma \circ \tau : F_0 \circ G_0 \Rightarrow F_1 \circ G_1 : C_0 \to C_2$ of two horizontally composable natural transformations is defined by pasting together naturality diagrams:

$$\begin{aligned} \sigma \circ \tau\,(o) &= \sigma \circ G_0(o) \cdot F_1 \circ \tau(o) = \boldsymbol{G_0(\sigma(o)) \cdot \tau(F_1(o))} \\ &= F_0 \circ \tau(o) \cdot \sigma \circ G_1(o) = \tau(F_0(o)) \cdot G_1(\sigma(o)) \end{aligned}$$

for any object $o \in Obj(C_0)$. The alternate definitions are equal, due to the naturality of τ. Diagram 3 illustrates horizontal composition.

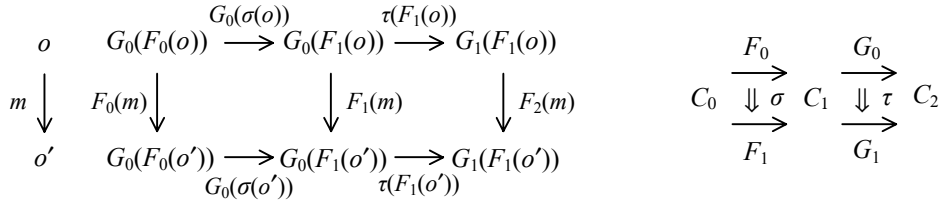


Diagram 3: Horizontal composition

- For any category C there is a *horizontal identity* natural transformation $1_C : id_C \Rightarrow id_C : C \to C$ defined in terms of identity morphisms in the category C: $1_C\,(o) = id^C_o$ for any object $o \in Obj(C)$. Here is the KIF representation for horizontal composition. We use the first alternative definition (in boldface).

```
(9) (CNG$function horizontal-composition)
    (CNG$signature horizontal-composition
        natural-transformation natural-transformation natural-transformation)

    (forall (?sigma (natural-transformation ?sigma)
```

```
             ?tau (natural-transformation ?tau))
        (<=> (exists (?n (natural-transformation ?n))
                 (= ?n (horizontal-composition ?sigma ?tau))))
             (= (target-category ?sigma) (source-category ?tau))))

    (forall (?sigma (natural-transformation ?sigma)
             ?tau (natural-transformation ?tau))
        (=> (= (target-category ?sigma) (source-category ?tau))
            (and (= (source-functor (horizontal-composition ?sigma ?tau))
                    (FUNC$composition (source-functor ?sigma) (source-functor ?tau)))
                 (= (target-functor (horizontal-composition ?sigma ?tau))
                    (FUNC$composition (target-functor ?sigma) (target-functor ?tau)))
                 (= (source-category (horizontal-composition ?sigma ?tau))
                    (source-category ?sigma))
                 (= (target-category (horizontal-composition ?sigma ?tau))
                    (target-category ?tau)))))

    (forall (?sigma (natural-transformation ?sigma)
             ?tau (natural-transformation ?tau))
        (=> (= (target-category ?sigma) (source-category ?tau))
            (= (component (horizontal-composition ?sigma ?tau))
               (SET.FTN$composition
                   ((SET.LIM.PBK$pairing
                        (CAT$composable-opspan (target-category ?sigma)))
                    (SET.FTN$composition
                        (component ?sigma)
                        (FUNC$morphism (source-functor ?tau)))
                    (SET.FTN$composition
                        (FUNC$object (target ?sigma))
                        (component ?tau))
                    (CAT$composition (target-category ?tau))))))))

(10) (CNG$function horizontal-identity)
     (CNG$signature horizontal-identity CAT$category natural-transformation)

     (forall (?c (CAT$category ?c))
         (and (= (source-functor (horizontal-identity ?c))
                 (FUNC$identity ?c))
              (= (target-functor (horizontal-identity ?c))
                 (FUNC$identity ?c))))

     (forall (?c (CAT$Category ?c) ?o ((CAT$object ?c) ?o))
         (= (component (horizontal-identity ?c))
            (FUNC$identity ?c)))
```

○ Given any category C and any category J, the *diagonal natural transformation* $\Delta_{J,\,C}$ maps a morphism $m : o_0 \to o_1$ to an associated constant natural transformation $\Delta_{J,\,C}(m) : \Delta_{J,\,C}(o_0) \Rightarrow \Delta_{J,\,C}(o_1) : J \to C$ between constant functors – its component function maps each object $j \in obj(J)$ to the morphism $m \in mor(C)$. This complete the definition of the diagonal functor $\Delta_{J,\,C} : C \to C^J$.

```
(11) (KIF$function diagonal)
     (KIF$signature diagonal CAT$category CAT$category CNG$function)
     (forall (?j (CAT$category ?j) ?c (CAT$category ?c))
         (and (CNG$signature (diagonal ?j ?c)
                  (CAT$morphism ?c) natural-transformation)
              (forall (?m ((CAT$morphism ?c) ?m))
                  (and (= (source-category ((diagonal ?j ?c) ?m)) ?j)
                       (= (target-category ((diagonal ?j ?c) ?m)) ?c)
                       (= (source-functor ((diagonal ?j ?c) ?m))
                          ((diagonal ?j ?c) ((CAT$source ?c) ?m)))
                       (= (target-functor ((diagonal ?j ?c) ?m))
                          ((diagonal ?j ?c) ((CAT$target ?c) ?m)))))))
```

Natural Transformation Theorems

- There is a quasi-category (a foundationally large category with object and morphism collections that are conglomerates), whose objects are functors, whose morphisms are natural transformation, whose source and target are the source and target functors for a natural transformation, whose composition is vertical composition, and whose identities are the vertical identities.
- There is a quasi-category, whose objects are categories, whose arrows are natural transformation, whose source and target are the source and target categories for a natural transformation, whose composition is horizontal composition, and whose identities are the horizontal identities.

We can prove the following theorems.

- The horizontal composition can be written in two different forms:

$$\sigma \circ \tau = (1_{F0} \circ \tau) \cdot (\sigma \circ 1_{G1}) = (\sigma \circ 1_{G0}) \cdot (1_{F1} \circ \tau).$$

- Vertical and horizontal composition satisfy the *interchange law* (Figure 2):

$$(\sigma_0 \cdot \sigma_1) \circ (\tau_0 \cdot \tau_1) = (\sigma_0 \circ \tau_0) \cdot (\sigma_1 \circ \tau_1)$$

for any three categories, six functors and four natural transformations as in the diagram on the right. Here is the KIF formalization of the interchange law.

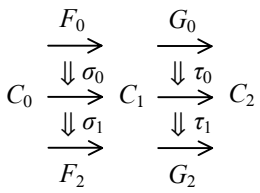


Figure 2: Interchange Law

```
(forall (?sigma0 (natural-transformation ?sigma0)
         ?sigma1 (natural-transformation ?sigma1)
         ?tau0 (natural-transformation ?tau0)
         ?tau1 (natural-transformation ?tau1))
    (=> (and (= (target-functor ?sigma0) (source-functor ?sigma1))
             (= (target-functor ?tau0) (source-functor ?tau1))
             (= (target-category ?sigma0) (source-category ?tau0)))
        (= (horizontal-composition
               (vertical-composition ?sigma0 ?sigma1)
               (vertical-composition ?tau0 ?tau1))
           (vertical-composition
               (horizontal-composition ?sigma0 ?tau0)
               (horizontal-composition ?sigma1 ?tau1)))))
```

The Namespace of Large Adjunctions

Adjunctions

ADJ

Categories are not only related by functors but also related through adjunctions.

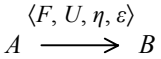


Figure 1: Adjunction

- An *adjunction* (Figure 1) $\langle F, U, \eta, \varepsilon \rangle : A \to B$ consists of a pair of natural transformations called the *unit* $\eta : Id_A \Rightarrow F \cdot U$ and *counit* $\varepsilon : U \cdot F \Rightarrow Id_B$ of the adjunction, a pair of functors called the *left adjoint* (or free functor) $F : A \to B$ and the *right adjoint* (or underlying functor) $U : B \to A$ of the adjunction, and a pair of categories called the *underlying category A* and *free category B* of the adjunction. An adjunction is governed by a pair of *triangle identities*,

  $$\eta F \bullet F\varepsilon = 1_F \qquad \text{and} \qquad \varepsilon U \bullet U\eta = 1_U,$$

  as illustrated in Diagram 1.

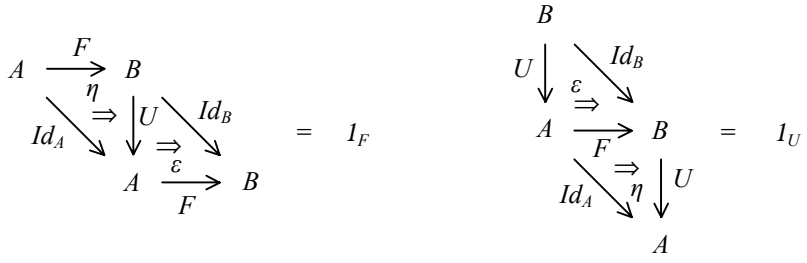


Diagram 1: Triangle identities

Here is the KIF representation for adjunctions. The conglomerate '`adjunction`' declared in axiom (1) represents abstract adjunctions, allowing one to *declare* adjunctions themselves. The functions '`underlying-category`' and '`free-category`' declared in axioms (2–3), represent the category aspect of adjunctions, allowing one to *declare* the underlying and free categories of adjunctions. The terms of axioms (4–7), which represent the functorial aspect of adjunctions by the functions '`left-adjoint`' and '`right-adjoint`' and the natural transformation aspect by the functions '`unit`' and '`counit`', resolve adjunctions into their parts. Axioms (8–9) represent the left-hand triangle and right-hand triangle equality in Diagram 1. Axiom (10) represents the fact that adjunctions are determined by their (underlying-category, free-category, left-adjoint, right-adjoint, unit, counit) sextuples.

```
(1) (CNG$conglomeration adjunction)

(2) (CNG$function underlying-category)
    (CNG$signature underlying-category adjunction CAT$category)

(3) (CNG$function free-category)
    (CNG$signature free-category adjunction CAT$category)

(4) (CNG$function left-adjoint)
    (CNG$signature left-adjoint adjunction FUNC$functor)
    (forall (?a (adjunction ?a))
        (and (= (CAT$source (left-adjoint ?a)) (underlying-category ?a))
             (= (CAT$target (left-adjoint ?a)) (free-category ?a))))

(5) (CNG$function right-adjoint)
    (CNG$signature right-adjoint adjunction FUNC$functor)
    (forall (?a (adjunction ?a))
        (and (= (CAT$source (right-adjoint ?a)) (free-category ?a))
             (= (CAT$target (right-adjoint ?a)) (underlying-category ?a))))

(6) (CNG$function unit)
    (CNG$signature unit adjunction NAT$natural-transformation)
```

```
        (forall (?a (adjunction ?a))
            (and (= (NAT$source (unit ?a))
                    (FUNC$identity (underlying-category ?a)))
                 (= (NAT$target (unit ?a))
                    (FUNC$composition (left-adjoint ?a) (right-adjoint ?a)))))

    (7) (CNG$function counit)
        (CNG$signature counit adjunction NAT$natural-transformation)
        (forall (?a (adjunction ?a))
            (and (= (NAT$source (counit ?a))
                    (FUNC$composition (right-adjoint ?a) (left-adjoint ?a)))
                 (= (NAT$target (counit ?a))
                    (FUNC$identity (free-category ?a)))))

    (8) (forall (?a (adjunction ?a))
            (= (NAT$vertical-composition
                   (NAT$horizontal-composition
                       (unit ?a) (NAT$vertical-identity (left-adjoint ?a)))
                   (NAT$horizontal-composition
                       (NAT$vertical-identity (left-adjoint ?a)) (counit ?a)))
               (NAT$vertical-identity (left-adjoint ?a)))))

    (9) (forall (?a (adjunction ?a))
            (= (NAT$vertical-composition
                   (NAT$horizontal-composition
                       (counit ?a) (NAT$vertical-identity (right-adjoint ?a)))
                   (NAT$horizontal-composition
                       (NAT$vertical-identity (right-adjoint ?a)) (unit ?a)))
               (NAT$vertical-identity (right-adjoint ?a)))))

    (10) (forall (?a1 (adjunction ?a1) ?a2 (adjunction ?a2))
             (=> (and (= (underlying-category ?a1) (underlying-category ?a2))
                      (= (free-category ?a1) (free-category ?a2))
                      (= (left-adjoint ?a1) (left-adjoint ?a2))
                      (= (right-adjoint ?a1) (right-adjoint ?a2))
                      (= (unit ?a1) (unit ?a2))
                      (= (counit ?a1) (counit ?a2)))
                 (= ?a1 ?a2)))
```

- Adjunctions and universal morphisms are closely related – we can prove the following theorem: if $U : B \to A$ is any functor, then U is the right adjoint functor in an adjunction $\langle F, U, \eta, \varepsilon \rangle : A \to B$ iff there is a universal morphism for every object $a \in obj(A)$. Given the adjunction, the universal morphism for $a \in obj(A)$ is given by $\langle \eta_a, F(a) \rangle$. Given the collection of universal morphisms $\{\langle m_a, \tilde{a} \rangle \mid a \in obj(A)\}$, define the object part of F by $F(a) = \tilde{a}$ and define the morphism part of F by $F(m : a \to a') $ is the unique morphism $\eta_a \cdot_A U(F(m)) = m \cdot \eta_{\hat{a}}$.

  Here is the KIF formalization of this theorem.

```
    (forall (?u (functor ?u))
        (<=> (exists (?adj (adjunction ?adj))
                 (= ?u (right-adjoint ?adj))
             (forall (?a ((CAT$object (target ?u)) ?a))
                 (exists (?x (((universal-morphism ?f) ?a) ?x)))))))
```

- It is a standard fact that every adjunction $\langle F, U, \eta, \varepsilon \rangle : A \to B$ gives rise to a monad $\langle T, \eta, \mu \rangle$ in a category A, where

$$T = F \circ U$$

$$\eta = \eta$$

$$\mu = 1_F \circ \varepsilon \circ 1_U$$

```
    (11) (CNG$function adjunction-monad)
         (CNG$signature adjunction-monad adjunction MND$monad)
         (forall (?a (adjunction ?a))
             (and (= (MND$underlying-category (adjunction-monad ?a))
                     (underlying-category ?a))
```

```
(= (MND$endofunctor (adjunction-monad ?a))
   (FUNC$composition (left-adjoint ?a) (right-adjoint ?a)))
(= (MND$unit (adjunction-monad ?a))
   (unit ?a))
(= (MND$multiplication (adjunction-monad ?a))
   (NAT$horizontal-composition
       (NAT$vertical-identity (left-adjoint ?a))
       (NAT$horizontal-composition
           (counit ?a)
           (NAT$vertical-identity (right-adjoint ?a)))))))
```

- Consider the opposite direction. We know that every monad $\langle T, \eta, \mu \rangle$ gives rise to two distinguished adjunctions, the Eilenberg-Moore adjunction $\langle F^M, U^M, \eta^M, \varepsilon^M \rangle : A^M \to A$ and the Kliesli adjunction $\langle F_M, U_M, \eta_M, \varepsilon_M \rangle : A_M \to A$. If that monad is the one generated by an adjunction $\langle F, U, \eta, \varepsilon \rangle : A \to B$, then the three adjunctions are comparable: there exists two distinguished functors, the *Kliesli comparison functor* $K_M : A_M \to B$ and the *Eilenberg-Moore comparison functor* $K^M : B \to A^M$, that satisfy the following identities.

$$\begin{aligned} U &= K^M \circ U^M \\ F^M &= F \circ K^M \\ U_M &= K_M \circ U \\ F &= F_M \circ K_M \end{aligned}$$

The Eilenberg-Moore comparison functor $K^M : B \to A^M$ maps an object $b \in obj(B)$ to the *free* algebra $K^M(b) = \langle U(b), U(\varepsilon(b)) \rangle$ and maps a B-morphism $h : b \to b'$ to the homomorphism $F^M(h) = U(h) : \langle U(b), U(\varepsilon(b)) \rangle \to \langle U(b'), U(\varepsilon(b')) \rangle$.

The Kliesli comparison functor $K_M : A_M \to B$ maps an object $a \in obj(A_M)$ to the B-object algebra $K^M(b) = F(a)$ and maps an A_M-morphism $\langle h, a' \rangle : a \to a'$, where $h : a \to T(a')$ is an A-morphism, to the *extension* B-morphism $F(h') \cdot_B \varepsilon(F(a')) : F(a) \to F(a')$.

```
(12) (CNG$function free)
     (CNG$signature free adjunction SET.FTN$function)
     (forall (?a (adjunction ?a))
         (and (= (SET.FTN$source (free ?a))
                 (CAT$object (underlying-category ?a)))
              (= (SET.FTN$target (free ?a))
                 (MND$algebra (adjunction-monad ?a)))
              (= (SET.FTN$composition
                     (free ?a)
                     (MND.ALG$underlying-object (adjunction-monad ?a)))
                 (FUNC$object (right-adjoint ?a)))
              (= (SET.FTN$composition
                     (free ?a)
                     (MND.ALG$structure-map (adjunction-monad ?a)))
                 (SET.FTN$composition
                     (NAT$component (counit ?a))
                     (FUNC$morphism (right-adjoint ?a))))))

(13) (CNG$function eilenberg-moore-comparison)
     (CNG$signature eilenberg-moore-comparison adjunction FUNC$functor)
     (forall (?a (adjunction ?a))
         (and (= (FUNC$source (eilenberg-moore-comparison ?a))
                 (free-category ?a))
              (= (FUNC$target (eilenberg-moore-comparison ?a))
                 (MND.ALG$eilenberg-moore (adjunction-monad ?a)))
              (= (FUNC$object (eilenberg-moore-comparison ?a))
                 (free ?a))
              (= (SET.FTN$composition
                     (FUNC$morphism (eilenberg-moore-comparison ?a))
                     (MND.ALG$underlying-morphism (adjunction-monad ?a)))
                 (FUNC$morphism (right-adjoint ?a))))))

(14) (CNG$function extension)
```

```
        (CNG$signature extension adjunction SET.FTN$function)
        (forall (?a (adjunction ?a))
            (and (= (SET.FTN$source (extension ?a))
                    (CAT$morphism (MND.ALG$kliesli (adjunction-monad ?a))))
                 (= (SET.FTN$target (extension ?a))
                    (CAT$morphism (free-category ?a)))
                 (= (extension ?a)
                    (SET.FTN$composition
                        (SET.LIM.PBK$pairing
                            (CAT$composable-opspan (MND$free-category ?a)))
                            (SET.FTN$composition
                                (SET.LIM.PBK$projection1
                                    (MND.ALG$kliesli-morphism-opspan ?m))
                                (FUNC$morphism (left-adjoint ?a)))
                            (SET.FTN$composition
                                (SET.LIM.PBK$projection2
                                    (MND.ALG$kliesli-morphism-opspan ?m))
                                (SET.FTN$composition
                                    (FUNC$object (left-adjoint ?a))
                                    (NAT$component (counit ?a)))))
                        (CAT$composition (free-category ?a))))))

    (15) (CNG$function kliesli-comparison)
         (CNG$signature kliesli-comparison adjunction FUNC$functor)
         (forall (?a (adjunction ?a))
             (and (= (FUNC$source (kliesli-comparison ?a))
                     (MND.ALG$kliesli (adjunction-monad ?a)))
                  (= (FUNC$target (kliesli-comparison ?a))
                     (free-category ?a))
                  (= (FUNC$object (kliesli-comparison ?a))
                     (FUNC$object (left-adjoint ?a)))
                  (= (FUNC$morphism (kliesli-comparison ?a))
                     (extension ?a))))
```

○ Given any two categories A and B, a (*strong*) *reflection* of A into B is an adjunction $\langle F, U, \eta, 1_{IdB} \rangle : A \rightarrow B$ whose counit is the identity, $\varepsilon = 1_{IdB}$, with $U \cdot F = Id_B$, so that U has injective object and morphism functions and B is a subcategory of A via U. That is, a subcategory $B \subseteq A$ is a *reflection* of A when the injection functor has a left adjoint right inverse (lari). Dually, given any two categories A and B, a (*strong*) *coreflection* of A into B is an adjunction $\langle F, U, 1_{IdA}, \varepsilon \rangle : A \rightarrow B$ whose unit is the identity, $\eta = 1_{IdA}$, with $F \cdot U = Id_A$, so that F has injective object and morphism functions and A is a subcategory of B via F. That is, a subcategory $A \subseteq B$ is a *coreflection* of B when the injection functor has a right adjoint right inverse (rari).

Here is the KIF formalization of for reflections and coreflections.

```
(16) (CNG$conglomeration reflection)
     (CNG$subconglomerate reflection adjunction)
     (forall (?a (adjunction ?a))
         (<=> (reflection ?a)
              (and (= (FUNC$composition (right-adjoint ?a) (left-adjoint ?a))
                      (FUNC$identity (free-category ?a)))
                   (= (counit ?a)
                      (NAT$vertical-identity (FUNC$identity (free-category ?a))))))))

(17) (CNG$conglomeration coreflection)
     (CNG$subconglomerate coreflection adjunction)
     (forall (?a (adjunction ?a))
         (<=> (coreflection ?a)
              (and (= (FUNC$composition (left-adjoint ?a) (right-adjoint ?a))
                      (FUNC$identity (underlying-category ?a)))
                   (= (unit ?a)
                      (NAT$vertical-identity
                          (FUNC$identity (underlying-category ?a))))))))
```

Adjunction Morphisms

ADJ.MOR

Adjunctions are related (vertically) by conjugate pairs of natural transformations.

- Suppose that two adjunctions $\langle F, U, \eta, \varepsilon \rangle, \langle F', U', \eta', \varepsilon' \rangle : A \to B$ share a common underlying (source) category A and a common free (target) category B. A *conjugate pair* of natural transformations $\langle \sigma, \tau \rangle : \langle F, U, \eta, \varepsilon \rangle \Rightarrow \langle F', U', \eta', \varepsilon' \rangle : A \to B$ from source adjunction $\langle F, U, \eta, \varepsilon \rangle$ to target functor $\langle F', U', \eta', \varepsilon' \rangle$, visualized 2-dimensionally in Figure 2, consists of a *left conjugate* natural transformation $\sigma : F \Rightarrow F'$ between the left adjoint (free) functors and a (contravariant) *right conjugate* natural transformation $\tau : U' \Rightarrow U$ between the right adjoint (underlying) functors, which satisfy either of the equivalent conditions in Table 1:

$$\langle F, U, \eta, \varepsilon \rangle \xrightarrow{\quad} A \;\Downarrow \langle \sigma, \tau \rangle\; B \xrightarrow{\quad} \langle F', U', \eta', \varepsilon' \rangle$$

Figure 2: Conjugate Pair

Table 1: Equivalent Conditions for Conjugate Pairs

τ	$=$	$U'\eta \bullet U'\sigma U \bullet \varepsilon' U$	σ	$=$	$\eta' F \bullet F'\tau F \bullet F'\varepsilon$
$\tau F \bullet \varepsilon$	$=$	$U'\sigma \bullet \varepsilon'$	$\eta \bullet \sigma U$	$=$	$\eta' \bullet F'\tau$

As these equivalents indicate, the natural transformation σ determines the natural transformation τ, and vice versa. Therefore, a conjugate pair is determined by either its left or right conjugate natural transformation. Here is the KIF formalization of conjugate pairs. Axiom (16) gives the left two of the above equivalent conditions.

```
(11) (CNG$conglomeration conjugate-pair)

(12) (CNG$function source)
     (CNG$signature source conjugate-pair ADJ$adjunction)

(13) (CNG$function target)
     (CNG$signature target conjugate-pair ADJ$adjunction)

     (forall (?p (conjugate-pair ?p))
         (and (= (ADJ$underlying-category (source ?p))
                 (ADJ$underlying-category (target ?p)))
              (= (ADJ$free-category (source ?p))
                 (ADJ$free-category (target ?p)))))

(14) (CNG$function left-conjugate)
     (CNG$signature left-conjugate conjugate-pair NAT$natural-transformation)

(15) (CNG$function right-conjugate)
     (CNG$signature right-conjugate conjugate-pair NAT$natural-transformation)

    (forall (?p (conjugate-pair ?p))
         (and (= (NAT$source (left-conjugate ?a))
                 (ADJ$left-adjoint (source ?p)))
              (= (NAT$target (left-conjugate ?a))
                 (ADJ$left-adjoint (target ?p)))
              (= (NAT$source (right-conjugate ?a))
                 (ADJ$right-adjoint (target ?p)))
              (= (NAT$target (right-conjugate ?a))
                 (ADJ$right-adjoint (source ?p)))))

(16) (forall (?p (conjugate-pair ?p))
         (and [τ = U'η • U'σU • ε'U]
              (= (left-conjugate ?a)
                 (ADJ$vertical-composition
                   (ADJ$vertical-composition
                     (ADJ$horizontal-composition
                       (NAT$vertical-identity (ADJ$right-adjoint (target ?p)))
                       (ADJ$unit (source ?p)))
                     (ADJ$horizontal-composition
                       (ADJ$horizontal-composition
```

```
                        (NAT$vertical-identity (ADJ$right-adjoint (target ?p)))
                        (left-conjugate ?p))
                      (NAT$vertical-identity (ADJ$right-adjoint (source ?p)))))
                  (ADJ$horizontal-composition
                    (ADJ$counit (target ?p))
                    (NAT$vertical-identity (ADJ$right-adjoint (source ?p))))))
            [τF • ε = U′σ • ε′]
            (= (ADJ$vertical-composition
                 (ADJ$horizontal-composition
                   (right-conjugate ?p)
                   (NAT$vertical-identity (ADJ$left-adjoint (source ?p))))
                 (ADJ$counit (source ?p)))
               (ADJ$vertical-composition
                 (ADJ$horizontal-composition
                   (NAT$vertical-identity (ADJ$right-adjoint (target ?p)))
                   (left-conjugate ?p))
                 (ADJ$counit (target ?p))))
    ))
```

Examples

Here are examples of adjunctions defined elsewhere, but asserted to be functors here.

- There is a natural transformation η from the identity functor on Classification to the composition of the underlying instance and instance power functors, whose component at any classification is the extent infomorphism associated with that classification. The underlying instance functor

  inst : Classification → $\mathrm{Set}^{\mathrm{op}}$

  is left adjoint *inst* ⊣ *pow* to the instance power functor

  pow : $\mathrm{Set}^{\mathrm{op}}$ → Classification,

  and η is the unit of this adjunction. Here is the KIF formalization for these facts, expressed in an external namespace.

```
(NAT$natural-transformation eta)
(= (NAT$source eta) (FUNC$identity Classification))
(= (NAT$target eta) (FUNC$composition [instance instance-power]))
(= (NAT$component eta) cls$extent)

(FUNC$composable [instance instance-power])
(= (FUNC$composition [instance-power instance]) (FUNC$identity set))

(ADJ$adjunction inst-pow)
(= (ADJ$underlying-category inst-pow) Classification)
(= (ADJ$free-category inst-pow)       Set)
(= (ADJ$left-adjoint inst-pow)        instance)
(= (ADJ$right-adjoint inst-pow)       instance-power)
(= (ADJ$unit inst-pow)                eta)
(= (ADJ$counit inst-pow)              (NAT$identity (FUNC$identity Set)))
```

The Namespace of Large Monads

Monads and Monad Morphisms

MND

In one sense, monads are universal algebra lifted to category theory. For any type of algebra, such as groups, complete semilattices, etcetra, there is its category of algebras Alg, its forgetful functor U to Set and its left adjoint free functor F in the reverse direction. The composite functor $T = F \circ U$ on Set comes equipped with two natural transformations that give it a monoid-like structure.

- A *monad* $\langle T, \eta, \mu \rangle$ on a category A is a triple consisting of an underlying endofunctor $T : A \to A$ and two natural transformations

$$\eta : Id_A \Rightarrow T \text{ and } \mu : T \circ T \Rightarrow T$$

which satisfy the following commuting diagrams (Table 1) (where $T^2 = T \circ T$ and $T^3 = T \circ T \circ T$):

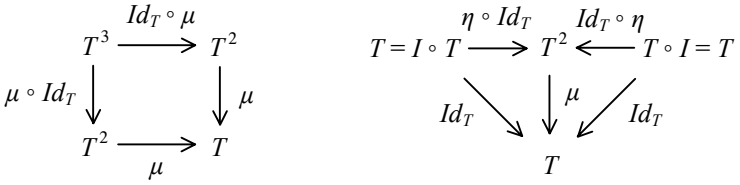


Table 1: Monad

The natural transformation is $\eta : Id_A \Rightarrow T$ called the *unit* of the monad, and the natural transformation $\mu : T \circ T \Rightarrow T$ is called the *multiplication* of the monad. In Table 1, the axiom on the left is called the *associative law* for the monad and the axioms on the right are called the *left unit law* and the *right unit law*, respectively.

Here is the KIF representation for monads. The conglomerate '`monad`' declared in axiom (1) represents the collection of monads, allowing one to *declare* monads. The function '`underlying-category`' declared in axiom (2) represents the underlying or base category of adjunctions. The terms of axioms (3–5), which represent the functorial and natural transformation aspects of monads by the functions '`underlying-functor`', '`unit`' and '`multiplication`', resolve adjunctions into their parts. Axiom (6) represents the associative law for adjunctions, and axioms (7) represent the left and right unit laws for adjunctions (Table 1). Axiom (8) represents the fact that monads are determined by their (underlying-functor, unit, multiplication) triples.

```
(1) (CNG$conglomerate monad)

(2) (CNG$function underlying-category)
    (CNG$signature underlying-category monad CAT$category)

(3) (CNG$function underlying-functor)
    (CNG$signature underlying-functor monad FUNC$functor)
    (forall (?m (monad ?m))
        (and (= (FUNC$source (underlying-functor ?m)) (underlying ?m))
             (= (FUNC$target (underlying-functor ?m)) (underlying ?m))))

(4) (CNG$function unit)
    (CNG$signature unit monad NAT$natural-transformation)
    (forall (?m (monad ?m))
        (and (= (FUNC$source-functor (unit ?m))
                (FUNC$identity (underlying-category ?m)))
             (= (FUNC$target-functor (unit ?m))
                (underlying-functor ?m))))

(5) (CNG$function multiplication)
    (CNG$signature multiplication monad NAT$natural-transformation)
    (forall (?m (monad ?m))
```

```
          (and (= (FUNC$source-functor (multiplication ?m))
                  (FUNC$composition (underlying-functor ?m) (underlying-functor ?m)))
               (= (FUNC$target-functor (multiplication ?m))
                  (underlying-functor ?m))))

(6) (forall (?m (monad ?m))
        (= (NAT$vertical-identity
               (NAT$horizontal-composition
                   (NAT$vertical-identity (underlying-functor ?m))
                   (multiplication ?m))
               (multiplication ?m))
           (NAT$vertical-identity
               (NAT$horizontal-composition
                   (multiplication ?m)
                   (NAT$vertical-identity (underlying-functor ?m)))
               (multiplication ?m))))

(7) (forall (?m (monad ?m))
        (and (= (NAT$vertical-identity
                    (NAT$horizontal-composition
                        (unit ?m)
                        (NAT$vertical-identity (underlying-functor ?m)))
                    (multiplication ?m))
                (NAT$vertical-identity (underlying-functor ?m)))
             (= (NAT$vertical-identity
                    (NAT$horizontal-composition
                        (NAT$vertical-identity (underlying-functor ?m))
                        (unit ?m))
                    (multiplication ?m))
                (NAT$vertical-identity (underlying-functor ?m))))))

(8) (forall (?m1 (monad ?m1) ?m2 (monad ?m2))
         (=> (and (= (underlying-functor ?m1) (underlying-functor ?m2))
                  (= (unit ?m1) (unit ?m2))
                  (= (multiplication ?m1) (multiplication ?m2)))
             (= ?m1 ?m2)))
```

○ Monads are related by their morphisms. A *morphism of monads* $\tau : \langle T, \eta, \mu \rangle \Rightarrow \langle T', \eta', \mu' \rangle$ is a natural transformation $\tau : T \Rightarrow T'$ which preserves multiplication and unit in the sense that the diagrams in Table 2 commute. In Table 2, the axiom on the left represents *preservation of multiplication* and the axiom on the right represents *preservation of unit*.

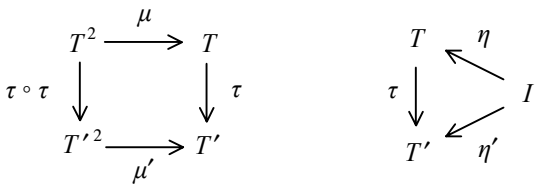


Table 2: Monad Morphism

```
(9) (CNG$conglomerate monad-morphism)

(10) (CNG$function source)
     (CNG$signature source monad-morphism monad)

(11) (CNG$function target)
     (CNG$signature source monad-morphism monad)

(12) (CNG$function underlying-natural-transformation)
     (CNG$signature underlying-natural-transformation
        monad-morphism NAT$natural-transformation)

(13) (forall (?t (monad-morphism ?t))
         (and (= (underlying-category (source ?t))
                 (underlying-category (target ?t)))
```

```
(= (NAT$source-functor (underlying-natural-transformation ?t))
   (underlying-functor (source ?t)))
(= (NAT$target-functor (underlying-natural-transformation ?t))
   (underlying-functor (target ?t)))))
(= (NAT$vertical-identity (multiplication (source ?t)) ?t)
   (NAT$vertical-identity
       (NAT$horizontal-composition ?t ?t)
       (multiplication (target ?t))))
(= (NAT$vertical-identity (unit (source ?t)) ?t)
   (unit (target ?t)))))
```

Algebras and Freeness

MND.ALG

For any monad $M = \langle T, \eta, \mu \rangle$ the Eilenberg-Moore category of algebras represents universal algebras and their homomorphisms.

- If $M = \langle T, \eta, \mu \rangle$ is a monad on category A, then an *M-algebra* $\langle a, \xi \rangle$ is a pair consisting of an object $a \in obj(A)$ (the *underlying object* of the algebra), and a morphism $\xi : T(a) \to a$ (called the *structure map* of the algebra) which makes the diagrams in Table 3 commute. The diagram on the left in Table 3 is called the *associative law* for the algebra and the diagram on the right is called the *unit law*.

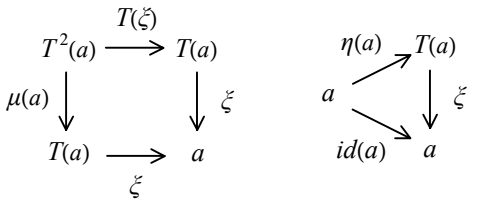


Table 3: Algebra

```
(1) (CNG$function algebra)
    (CNG$signature algebra MND$monad SET.class)

(2) (CNG$function underlying-object)
    (CNG$signature underlying-object MND$monad SET.FTN.function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (underlying-object ?m))
                (algebra ?m))
             (= (SET.FTN$target (underlying-object ?m))
                (CAT$object (MND$underlying-category ?m)))))

(3) (CNG$function structure-map)
    (CNG$signature structure-map MND$monad SET.FTN.function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (structure-map ?m))
                (algebra ?m))
             (= (SET.FTN$target (structure-map ?m))
                (CAT$morphism (MND$underlying-category ?m)))
             (= (SET.FTN$composition
                    (structure-map ?m)
                    (CAT$source (MND$underlying-category ?m)))
                (SET.FTN$composition
                    (underlying-object ?m)
                    (FUNC$object (MND$underlying-functor ?m))))
             (= (SET.FTN$composition
                    (structure-map ?m)
                    (CAT$target (MND$underlying-category ?m)))
                (underlying-object ?m))
             (forall (?a ((algebra ?m) ?a))
                 (and (= ((CAT$composition (MND$underlying-category ?m))
                             [(FUNC$morphism (MND$underlying-functor ?m))
                                 ((structure-map ?m) ?a))
                              ((structure-map ?m) ?a)])
                         ((CAT$composition (MND$underlying-category ?m))
                             [((NAT$component (MND$multiplication ?m))
```

```
                        ((underlying-object ?m) ?a))
                    ((structure-map ?m) ?a)]))
            (= ((CAT$composition (MND$underlying-category ?m))
                   [((NAT$component (MND$unit ?m))
                        ((underlying-object ?m) ?a))
                    ((structure-map ?m) ?a)]))
               ((CAT$identity (MND$underlying-category ?m))
                   ((underlying-object ?m) ?a)))))))
```

```
(4) (CNG$function algebra-pair)
    (CNG$signature algebra-pair MND$monad SET.LIM.PRD$diagram)
    (forall (?m (MND$monad ?m))
        (and (= (SET.LIM.PRD$class1 (algebra-pair ?m)) (algebra ?m))
             (= (SET.LIM.PRD$class2 (algebra-pair ?m)) (algebra ?m))))
```

○ If $M = \langle T, \eta, \mu \rangle$ is a monad on category A, an *M-homomorphism* $h : \langle a, \xi \rangle \to \langle a', \xi' \rangle$ is an A-morphism $h : a \to a'$ between the underlying objects that preserves the algebraic structure by satisfying the commutative diagram in Table 4.

$$\begin{array}{ccc} & \xi & \\ T(a) & \longrightarrow & a \\ {\scriptstyle T(h)}\downarrow & & \downarrow{\scriptstyle h} \\ T(a') & \longrightarrow & a' \\ & \xi' & \end{array}$$

Table 4: Homomorphism

```
(5) (CNG$function homomorphism)
    (CNG$signature homomorphism MND$monad SET.class)

(6) (CNG$function source)
    (CNG$signature source MND$monad SET.FTN.function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (source ?m))
                (homomorphism ?m))
             (= (SET.FTN$target (source ?m))
                (algebra ?m))))

(7) (CNG$function target)
    (CNG$signature target MND$monad SET.FTN.function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (target ?m))
                (homomorphism ?m))
             (= (SET.FTN$target (target ?m))
                (algebra ?m))))

(8) (CNG$function underlying-morphism)
    (CNG$signature underlying-morphism MND$monad SET.FTN.function)
    (forall (?m (MND$monad ?m))
        (and (= (SET.FTN$source (underlying-morphism ?m))
                (homomorphism ?m))
             (= (SET.FTN$target (underlying-morphism ?m))
                (CAT$morphism (MND$underlying-category ?m)))
             (= (SET.FTN$composition
                    (underlying-morphism ?m)
                    (CAT$source (MND$underlying-category ?m)))
                (SET.FTN$composition (source ?m) (underlying-object ?m)))
             (= (SET.FTN$composition
                    (underlying-morphism ?m)
                    (CAT$target (MND$underlying-category ?m)))
                (SET.FTN$composition (target ?m) (underlying-object ?m)))
             (forall (?h ((homomorphism ?m) ?h))
                 (= ((CAT$composition (MND$underlying-category ?m))
                        [((structure-map ?m) (source ?h)) ?h])
                    ((CAT$composition (MND$underlying-category ?m))
                        [((FUNC$morphism (underlying-functor ?m)) ?h)
```

```
                        ((structure-map ?m) (target ?h))])))))

(9) (CNG$function composable-opspan)
    (CNG$signature composable-opspan MND$monad SET.LIM.PBK$diagram)
    (forall (?m (MND$monad ?m))
        (and (= (SET.LIM.PRD$class1 (composable-opspan ?m)) (homomorphism ?m))
             (= (SET.LIM.PRD$class2 (composable-opspan ?m)) (homomorphism ?m))
             (= (SET.LIM.PRD$opvertex (composable-opspan ?m)) (algebra ?m))
             (= (SET.LIM.PRD$opfirst (composable-opspan ?m)) (target ?m))
             (= (SET.LIM.PRD$opsecond (composable-opspan ?m)) (source ?m))))

(10) (CNG$function composable)
     (CNG$signature composable MND$monad SET$class)
     (forall (?m (MND$monad ?m))
         (= (composable ?m)
            (SET.LIM.PBK$pullback (composable-opspan ?m))))

(11) (CNG$function composition)
     (CNG$signature composition MND$monad SET.FTN.function)
     (forall (?m (MND$monad ?m))
         (and (= (SET.FTN$source (composition ?m))
                 (composable ?m))
              (= (SET.FTN$target (composition ?m))
                 (homomorphism ?m))
              (forall (?h1 ?h2 ((composable ?m) [?h1 ?h2]))
                  (= (underlying-morphism ((composition ?m) [?h1 ?h2]))
                     ((CAT$composition (MND$underlying-category ?m))
                         [(underlying-morphism ?h1) (underlying-morphism ?h2)])))))

(12) (CNG$function identity)
     (CNG$signature identity MND$monad SET.FTN.function)
     (forall (?m (MND$monad ?m))
         (and (= (SET.FTN$source (identity ?m))
                 (algebra ?m))
              (= (SET.FTN$target (identity ?m))
                 (homomorphism ?m))
              (forall (?a ((algebra ?m) ?a))
                  (= (underlying-morphism ((identity ?m) ?a))
                     ((CAT$identity (MND$underlying-category ?m))
                         ((underlying-object ?m) ?a)))))))
```

○ For any monad $M = \langle T, \eta, \mu \rangle$ on category A, the M-algebras and M-homomorphisms form the *Eilenberg-Moore category* A^M.

```
(13) (CNG$function eilenberg-moore)
     (CNG$signature eilenberg-moore MND$monad CAT$category)
     (forall (?m (monad ?m))
         (and (= (CAT$object (eilenberg-moore ?m))
                 (algebra ?m))
              (= (CAT$morphism (eilenberg-moore ?m))
                 (homomorphism ?m))
              (= (CAT$source (eilenberg-moore ?m))
                 (source ?m))
              (= (CAT$target (eilenberg-moore ?m))
                 (target ?m))
              (= (CAT$composition (eilenberg-moore ?m))
                 (composition ?m))
              (= (CAT$identity (eilenberg-moore ?m))
                 (identity ?m)))))
```

○ For any monad $M = \langle T, \eta, \mu \rangle$ on category A, there is an *underlying* functor $U^M : A^M \to A$, a *free* functor $F^M : A \to A^M$ that maps an object $a \in obj(A)$ to the "free" algebra $\langle a, \mu(a) \rangle$ and maps an A-morphism $h : a \to a'$ to the homomorphism $F^M(h) = T(h) : \langle a, \mu(a) \rangle \to \langle a', \mu(a') \rangle$, a *unit* natural transformation $\eta^M : Id_A \Rightarrow F^M \circ U^M$ with component $\eta^M(a) = \eta(a)$ at any object $a \in obj(A)$, and a *counit* natural transformation $\varepsilon^M : U^M \circ F^M \Rightarrow Id_A$ with component $\varepsilon^M(\langle a, \xi \rangle) = \xi$ at any algebra $\langle a, \xi \rangle$. This data forms an *adjunction* $\langle F^M, U^M, \eta^M, \varepsilon^M \rangle : A \to A^M$.

```
(14) (CNG$function underlying-eilenberg-moore)
     (CNG$signature underlying-eilenberg-moore MND$monad FUNC$functor)
```

```
        (forall (?m (MND$monad ?m))
            (and (= (FUNC$source (underlying-eilenberg-moore ?m))
                    (eilenberg-moore ?m))
                 (= (FUNC$target (underlying-eilenberg-moore ?m))
                    (MND$underlying-category ?m))
                 (= (FUNC$object (underlying-eilenberg-moore ?m))
                    (underlying-object ?m))
                 (= (FUNC$morphism (underlying-eilenberg-moore ?m))
                    (underlying-morphism ?m))))

    (15) (CNG$function free-eilenberg-moore)
         (CNG$signature free-eilenberg-moore MND$monad FUNC$functor)
         (forall (?m (MND$monad ?m))
             (and (= (FUNC$source (free-eilenberg-moore ?m))
                     (MND$underlying-category ?m))
                  (= (FUNC$target (free-eilenberg-moore ?m))
                     (eilenberg-moore ?m))
                  (= (SET.FTN$composition
                         (FUNC$object (free-eilenberg-moore ?m))
                         (underlying-object ?m))
                     (FUNC$object (MND$underlying-functor ?m)))
                  (= (SET.FTN$composition
                         (FUNC$object (free-eilenberg-moore ?m))
                         (structure-map ?m))
                     (NAT$component (MND$multiplication ?m)))
                  (= (SET.FTN$composition
                         (FUNC$morphism (free-eilenberg-moore ?m))
                         (underlying-morphism ?m))
                     (FUNC$morphism (MND$underlying-functor ?m)))))

    (16) (CNG$function unit-eilenberg-moore)
         (CNG$signature unit-eilenberg-moore MND$monad NAT$natural-transformation)
         (forall (?m (MND$monad ?m))
             (and (= (NAT$source-functor (unit-eilenberg-moore ?m))
                     (FUNC$identity (MND$underlying-category ?m)))
                  (= (NAT$target-functor (unit-eilenberg-moore ?m))
                     (FUNC$composition
                         (free-eilenberg-moore ?m)
                         (underlying-eilenberg-moore ?m)))
                  (= (NAT$component (unit-eilenberg-moore ?m))
                     (NAT$component (MND$unit ?m)))))

    (17) (CNG$function counit-eilenberg-moore)
         (CNG$signature counit-eilenberg-moore MND$monad NAT$natural-transformation)
         (forall (?m (MND$monad ?m))
             (and (= (NAT$source-functor (counit-eilenberg-moore ?m))
                     (FUNC$composition
                         (underlying-eilenberg-moore ?m)
                         (free-eilenberg-moore ?m)))
                  (= (NAT$target-functor (counit-eilenberg-moore ?m))
                     (FUNC$identity (MND$underlying-category ?m)))
                  (= (NAT$component (counit-eilenberg-moore ?m))
                     (structure-map ?m)))))

    (18) (CNG$function adjunction-eilenberg-moore)
         (CNG$signature adjunction-eilenberg-moore MND$monad NAT$natural-transformation)
         (forall (?m (MND$monad ?m))
             (and (= (NAT$underlying-functor (adjunction-eilenberg-moore ?m))
                     (underlying-eilenberg-moore ?m))
                  (= (NAT$free-functor (adjunction-eilenberg-moore ?m))
                     (free-eilenberg-moore ?m))
                  (= (NAT$unit (adjunction-eilenberg-moore ?m))
                     (unit-eilenberg-moore ?m))
                  (= (NAT$counit (adjunction-eilenberg-moore ?m))
                     (counit-eilenberg-moore ?m)))))
```

- We can then prove the theorem that the monad generated by the Eilenberg-Moore adjunction is the original monad.

```
        (forall (?m (MND$monad ?m))
            (= (ADJ$adjunction-monad (adjunction-eilenberg-moore ?m)) ?m))
```

For any monad $M = \langle T, \eta, \mu \rangle$ the *Kliesli category* represents the free part of universal algebras.

- Let $M = \langle T, \eta, \mu \rangle$ be a monad on category A. Restrict attention to the morphisms in A of the form $h : a \to T(a')$. The Kliesli category A_M has this as a morphism with source object $a \in obj(A_M)$ and target object $a' \in obj(A_M)$. So $obj(A_M) = obj(A)$, $mor(A_M) \subseteq mor(A)$, composition of two morphisms $h : a \to T(a')$ and $h' : a' \to T(a'')$ is defined by $h \cdot_{AM} h' = h \cdot_A h'^{\#}$ where $h'^{\#} = T(h') \cdot_A \mu(a'')$ is the *extension* operator, and the identity at an object $a \in obj(A_M)$ is $\eta(a) : a \to T(a)$. More precisely, as can be seen below, a morphism is a pair algebra $\langle h, a' \rangle$, where $h : a \to T(a')$ is an A-morphism. Clearly, foundational pullback opspans, cocones and pairing play a key role in the definition of the Kliesli category.

```
(19) (CNG$function kliesli-morphism-opspan)
     (CNG$signature kliesli-morphism-opspan MND$monad SET.LIM.PBK$diagram)
     (forall (?m (MND$monad ?m))
         (and (= (SET.LIM.PBK$class1 (kliesli-morphism-opspan ?m))
                 (CAT$morphism (MND$underlying-category ?m)))
              (= (SET.LIM.PBK$class2 (kliesli-morphism-opspan ?m))
                 (CAT$object (MND$underlying-category ?m)))
              (= (SET.LIM.PBK$opvertex (kliesli-morphism-opspan ?m))
                 (CAT$object (MND$underlying-category ?m)))
              (= (SET.LIM.PBK$opfirst (kliesli-morphism-opspan ?m))
                 (CAT$target (MND$underlying-category ?m)))
              (= (SET.LIM.PBK$opsecond (kliesli-morphism-opspan ?m))
                 (FUNC$object (MND$underlying-functor ?m)))))

(20) (CNG$function kliesli-identity-cone)
     (CNG$signature kliesli-identity-cone MND$monad SET.LIM.PBK$cocone)
     (forall (?m (MND$monad ?m))
         (and (= (SET.LIM.PBK$cone-diagram (kliesli-identity-cone ?m))
                 (Kliesli-morphism-opspan ?m))
              (= (SET.LIM.PBK$vertex (kliesli-identity-cone ?m))
                 (CAT$object (MND$underlying-category ?m)))
              (= (SET.LIM.PBK$first (kliesli-identity-cone ?m))
                 (NAT$component (MND$unit ?m)))
              (= (SET.LIM.PBK$second (kliesli-identity-cone ?m))
                 (SET.FTN$identity (MND$underlying-category ?m)))))

(21) (CNG$function kliesli-composable-opspan)
     (CNG$signature kliesli-composable-opspan MND$monad SET.LIM.PBK$diagram)
     (forall (?m (MND$monad ?m))
         (and (= (SET.LIM.PBK$class1 (kliesli-composable-opspan ?m))
                 (SET.LIM.PBK$pullback (kliesli-morphism-opspan ?m)))
              (= (SET.LIM.PBK$class2 (kliesli-composable-opspan ?m))
                 (SET.LIM.PBK$pullback (kliesli-morphism-opspan ?m)))
              (= (SET.LIM.PBK$opvertex (kliesli-composable-opspan ?m))
                 (CAT$object (MND$underlying-category ?m)))
              (= (SET.LIM.PBK$opfirst (kliesli-composable-opspan ?m))
                 (SET.LIM.PBK$projection2 (kliesli-morphism-opspan ?m)))
              (= (SET.LIM.PBK$opsecond (kliesli-composable-opspan ?m))
                 (SET.FTN$composition
                     (SET.LIM.PBK$projection1 (kliesli-morphism-opspan ?m))
                     (CAT$source (MND$underlying-category ?m))))))

(22) (CNG$function extension)
     (CNG$signature extension MND$monad SET.FTN$function)
     (forall (?m (MND$monad ?m))
         (and (= (SET.FTN$source (extension ?m))
                 (SET.LIM.PBK$pullback (kliesli-morphism-opspan ?m)))
              (= (SET.FTN$target (extension ?m))
                 (CAT$morphism (MND$underlying-category ?m)))
              (= (extension ?m)
                 (SET.FTN$composition
                     (SET.LIM.PBK$pairing
                         (CAT$composable-opspan (MND$underlying-category ?m)))
                         (SET.FTN$composition
                             (SET.LIM.PBK$projection1 (kliesli-morphism-opspan ?m))
                             (FUNC$morphism (MND$underlying-functor ?m)))
                         (SET.FTN$composition
                             (SET.LIM.PBK$projection2 (kliesli-morphism-opspan ?m))
```

```
                        (NAT$component (MND$multiplication ?m))))
                    (CAT$composition (MND$underlying-category ?m))))))

(23) (CNG$function kliesli)
     (CNG$signature kliesli MND$monad CAT$category)
     (forall (?m (monad ?m))
         (and (= (CAT$object (kliesli ?m))
                 (CAT$object (MND$underlying-category ?m)))
              (= ((CAT$morphism (kliesli ?m))
                 (SET.LIM.PBK$pullback (kliesli-morphism-opspan ?m)))
              (= (CAT$source (kliesli ?m))
                 (SET.FTN$composition
                     (SET.LIM.PBK$projection1 (kliesli-morphism-opspan ?m))
                     (CAT$source (MND$underlying-category ?m))))
              (= (CAT$target (kliesli ?m))
                 (SET.LIM.PBK$projection2 (kliesli-morphism-opspan ?m)))
              (= (CAT$composable (kliesli ?m))
                 (SET.LIM.PBK$pullback (kliesli-composable-opspan ?m)))
              (= (CAT$composition (kliesli ?m))
                 (SET.FTN$composition
                     (SET.LIM.PBK$pairing
                         (CAT$composable-opspan (MND$underlying-category ?m)))
                         (SET.FTN$composition
                             (SET.LIM.PBK$projection1 (kliesli-composable-opspan ?m))
                             (SET.LIM.PBK$projection1 (kliesli-morphism-opspan ?m)))
                         (SET.FTN$composition
                             (SET.LIM.PBK$projection2 (kliesli-composable-opspan ?m))
                             (SET.FTN$composition
                                 (SET.LIM.PBK$projection1
                                     (kliesli-morphism-opspan ?m)))
                                 (extension ?m))))
                     (CAT$composition (MND$underlying-category ?m))))
              (= (CAT$identity (kliesli ?m))
                 (SET.LIM.PBK$mediator (kliesli-identity-cone ?m)))))
```

- For any monad $M = \langle T, \eta, \mu \rangle$ on category A, there is an *underlying* functor $U_M : A_M \to A$ that maps an object $a \in obj(A_M)$ to the object $T(a) \in obj(A)$ and maps a morphism $\langle h, a' \rangle : a \to a'$ in A_M to the extension morphism $h^{\#} : T(a) \to T(a')$ in A, a *free* functor $F_M : A \to A_M$ that is the identity on objects and maps a A-morphism $h : a \to a'$ to the *embedded* morphism $\langle h \cdot \eta(a'), a' \rangle : a \to a'$ in A_M, a *unit* natural transformation $\eta_M : Id_A \Rightarrow F_M \circ U_M$ with component $\eta_M(a) = \eta(a)$ at any object $a \in obj(A)$, and a *counit* natural transformation $\varepsilon_M : U_M \circ F_M \Rightarrow Id_A$ with component $\varepsilon_M(a) = \langle id_A(T(a)), a \rangle : T(a) \to a$ at any $a \in obj(A_M)$. This data forms an *adjunction* $\langle F_M, U_M, \eta_M, \varepsilon_M \rangle : A \to A_M$.

```
(24) (CNG$function underlying-kliesli)
     (CNG$signature underlying-kliesli MND$monad FUNC$functor)
     (forall (?m (MND$monad ?m))
         (and (= (FUNC$source (underlying-kliesli ?m))
                 (kliesli ?m))
              (= (FUNC$target (underlying-kliesli ?m))
                 (MND$underlying-category ?m))
              (= (FUNC$object (underlying-kliesli ?m))
                 (FUNC$object (MND$underlying-functor ?m)))
              (= (FUNC$morphism (underlying-kliesli ?m))
                 (extension ?m)))))

(25) (CNG$function embed)
     (CNG$signature embed MND$monad SET.FTN$function)
     (forall (?m (MND$monad ?m))
         (and (= (SET.FTN$source (embed ?m))
                 (CAT$morphism (MND$underlying-category ?m)))
              (= (SET.FTN$target (embed ?m))
                 (CAT$morphism (kliesli ?m)))
              (= (SET.FTN$composition
                     (embed ?m)
                     (SET.LIM.PBK$projection1 (kliesli-morphism-opspan ?m)))
                 (SET.FTN$composition
                     (SET.LIM.PBK$pairing
                         (CAT$composable-opspan (MND$underlying-category ?m)))
                         (SET.FTN$identity
```

```
                            (CAT$morphism (MND$underlying-category ?m)))
                        (SET.FTN$composition
                            (CAT$target (MND$underlying-category ?m))
                            (NAT$component (MND$unit ?m))))
                    (CAT$composition (MND$underlying-category ?m))))
            (= (SET.FTN$composition
                    (embed ?m)
                    (SET.LIM.PBK$projection2 (kliesli-morphism-opspan ?m)))
               (CAT$target (MND$underlying-category ?m)))))

(26) (CNG$function free-kliesli)
     (CNG$signature free-kliesli MND$monad FUNC$functor)
     (forall (?m (MND$monad ?m))
         (and (= (FUNC$source (free-kliesli ?m))
                 (MND$underlying-category ?m))
              (= (FUNC$target (free-kliesli ?m))
                 (kliesli ?m))
              (= (FUNC$object (free-kliesli ?m))
                 (SET.FTN$identity (CAT$object (MND$underlying-category ?m))))
              (= (FUNC$morphism (free-kliesli ?m))
                 (embed ?m)))))

(27) (CNG$function unit-kliesli)
     (CNG$signature unit-kliesli MND$monad NAT$natural-transformation)
     (forall (?m (MND$monad ?m))
         (and (= (NAT$source-functor (unit-kliesli ?m))
                 (FUNC$identity (MND$underlying-category ?m)))
              (= (NAT$target-functor (unit-kliesli ?m))
                 (FUNC$composition
                     (free-kliesli ?m)
                     (underlying-kliesli ?m)))
              (= (NAT$component (unit-kliesli ?m))
                 (NAT$component (MND$unit ?m))))))

(28) (CNG$function counit-kliesli)
     (CNG$signature counit-kliesli MND$monad NAT$natural-transformation)
     (forall (?m (MND$monad ?m))
         (and (= (NAT$source-functor (counit-kliesli ?m))
                 (FUNC$composition (underlying-kliesli ?m) (free-kliesli ?m)))
              (= (NAT$target-functor (counit-kliesli ?m))
                 (FUNC$identity (MND$underlying-category ?m)))
              (= (NAT$component (counit-kliesli ?m))
                 (SET.FTN$composition
                     (FUNC$object (MND$underlying-functor ?m))
                     (CAT$identity (MND$underlying-category ?m)))))))

(29) (CNG$function adjunction-kliesli)
     (CNG$signature adjunction-kliesli MND$monad NAT$natural-transformation)
     (forall (?m (MND$monad ?m))
         (and (= (NAT$underlying-functor (adjunction-kliesli ?m))
                 (underlying-kliesli ?m))
              (= (NAT$free-functor (adjunction-kliesli ?m))
                 (free-kliesli ?m))
              (= (NAT$unit (adjunction-kliesli ?m))
                 (unit-kliesli ?m))
              (= (NAT$counit (adjunction-kliesli ?m))
                 (counit-kliesli ?m)))))
```

- We can then prove the theorem that the monad generated by the Kliesli adjunction is the original monad.

```
     (forall (?m (MND$monad ?m))
         (= (ADJ$adjunction-monad (adjunction-kliesli ?m)) ?m))
```

The Namespace of Colimits/Limits

COL

Colimits

The Information Flow Framework advances the following proposal: to relate ontologies via morphisms and to compose them using colimits. This section provides the foundation for that proposal. Colimits are important for manipulating and composing ontologies expressed in the object language. The use of colimits advocates a "building blocks approach" for ontology construction. Continuing the metaphor, this approach understands that the mortar between the ontological blocks must be strong and resilient in order to adequately support the ontological building, and requests that methods for composing component ontologies, such as merging, mapping and aligning ontologies, be made very explicit so that they can be analyzed. The Specware system of the Kestrel Institute, which is based on category theory, supports creation and combination of specifications (ontology analogs) using colimits. A compact but detailed discussion of classifications and infomorphisms with applications to this building blocks approach for ontology construction is given in the 6th ISKO paper The Information Flow Foundation for Conceptual Knowledge Organization.

Colimits form a separate namespace in the Category Theory Ontology. Within the colimit namespace are several subnamespaces concerned with binary coproducts, coequalizers and pushouts. To completely express colimits, we need elements from all the basic components of category theory – categories, functors, natural transformations and adjunctions. As such, colimits provide a glimpse of the other parts of the Category Theory Ontology – the other basic components used in colimits are indicated by the namespace prefixes in the KIF formalism.

Finite Colimits

For convenience of representation it is common to use special terminology for a few particular kinds of finite colimits: initial objects, binary coproducts, coequalizers and pushouts. Please note the very common formalisms that encode these. This commonality is expressed in the generalized colimits discussed after words.

Initial Objects

- Given a category C, an *initial object* $0_C \in obj(C)$ is an object (Figure 1) such that for any object $o \in obj(C)$ there is a *counique morphism* $!_{C,o} : 0_C \to o$. In the following KIF the term '`initial`' of axiom (1) represents the class of initial objects (possibly empty), and the term '`counique`' of axiom (2) represents the counique function. From the more general theorem that "all colimits for a particular diagram in a category are isomorphic," we can derive the fact that all initial objects are isomorphic. phic. We use a KIF definite description to define the counique function.

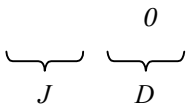


Figure 1: Initial object

```
(1)(CNG$function initial)
   (CNG$signature initial CAT$category SET.class)
   (forall (?c (CAT$category ?c))
       (SET$subclass (initial ?c) (CAT$object ?c))

(2)(CNG$function counique)
   (CNG$signature counique CAT$category SET.FTN$function)
   (forall (?c (CAT$category ?c))
       (= (counique ?c)
          (the (?f (SET.FTN$function ?f))
              (and (= (SET.FTN$source ?f) (CAT$object ?c))
                   (= (SET.FTN$target ?f) (CAT$morphism ?c))
                   (= (SET.FTN$composition ?f (CAT$source ?c))
                      ((SET.TOP$constant (CAT$object ?c) (CAT$object ?c))
                          (initial ?c)))
                   (= (SET.FTN$composition ?f (CAT$target ?c))
                      (SET.FTN$identity (CAT$object ?c)))))))
```

Binary Coproducts

COL.COPRD

A *binary coproduct* in a category *C* (Figure 2) is a finite colimit for a diagram of shape *set2* = • •. Such a diagram (of *C*-objects and *C*-morphisms) is called a *copair*.

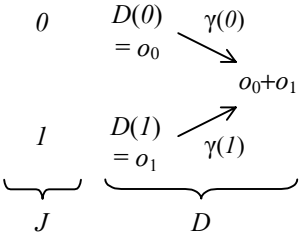


Figure 2: Binary coproduct

- A *copair* (of *C*-objects) is the appropriate base diagram for a binary coproduct in *C*. Each copair consists of a pair of *C*-objects called *object1* and *object2*. Let either 'diagram' or 'copair' be the COL.COPRD namespace term that denotes the *Copair* collection. Copairs are determined by their two component C-objects.

```
(1) (CNG$function diagram)
    (CNG$function copair)
    (= copair diagram)
    (CNG$signature diagram CAT$category SET$class)

(2) (CNG$function object1)
    (CNG$signature object1 CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (object1 ?c)) (diagram ?c))
             (= (SET.FTN$target (object1 ?c)) (CAT$object ?c))))

(3) (CNG$function object2)
    (CNG$signature object2 CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (object2 ?c)) (diagram ?c))
             (= (SET.FTN$target (object2 ?c)) (CAT$object ?c))))

    (forall (?c (CAT$category ?c)
             ?p1 ((diagram ?c) ?p1) ?p2 ((diagram ?c) ?p2))
        (=> (and (= ((object1 ?c) ?p1) ((object1 ?c) ?p2))
                 (= ((object2 ?c) ?p1) ((object2 ?c) ?p2)))
            (= ?p1 ?p2)))
```

- Every pair has an opposite.

```
(4) (CNG$function opposite)
    (CNG$signature opposite CAT$category SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opposite ?c)) (diagram ?c))
             (= (SET.FTN$target (opposite ?c)) (diagram ?c))
             (= (SET.FTN$composition (opposite ?c) (object1 ?c))
                (object2 ?c))
             (= (SET.FTN$composition (opposite ?c) (object2 ?c))
                (object1 ?c))))
```

- The opposite of the opposite is the original pair – the following theorem can be proven.

```
(forall (?c (CAT$category ?c))
    (= (SET.FTN$composition (opposite ?c) (opposite ?c))
       (SET.FTN$identity (diagram ?c))))
```

- *Coproduct cocones* are used to specify and axiomatize binary coproducts in a category *C*. Each coproduct cocone has an underlying coproduct *diagram* (copair), an *opvertex C*-object, and a pair of *C*-morphisms called *opfirst* and *opsecond*, whose common target *C*-object is the opvertex and whose source *C*-objects are the component *C*-objects in the underlying diagram. The opfirst and opsecond morphisms are the components of a natural transformation. A coproduct cocone is the very special case of a general colimit cocone over a coproduct diagram. Let 'cocone' be the COL.PSH namespace term that denotes the *Coproduct Cocone* collection.

```
(5) (CNG$function cocone)
    (CNG$signature cocone CAT$category SET$class)

(6) (CNG$function cocone-diagram)
```

```
    (CNG$signature cocone-diagram CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (cocone-diagram ?c)) (cocone ?c))
             (= (SET.FTN$target (cocone-diagram ?c)) (diagram ?c))))

(7) (CNG$function opvertex)
    (CNG$signature opvertex CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opvertex ?c)) (cocone ?c))
             (= (SET.FTN$target (opvertex ?c)) (CAT$object ?c))))

(8) (CNG$function opfirst)
    (CNG$signature opfirst CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opfirst ?c)) (cocone ?c))
             (= (SET.FTN$target (opfirst ?c)) (CAT$morphism ?c))
             (= (SET.FTN$composition (opfirst ?c) (CAT$source ?c))
                (SET.FTN$composition (cocone-diagram ?c) (object1 ?c)))
             (= (SET.FTN$composition (opfirst ?c) (CAT$target ?c))
                (opvertex ?c))))

(9) (CNG$function opsecond)
    (CNG$signature opsecond CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opsecond ?c)) (cocone ?c))
             (= (SET.FTN$target (opsecond ?c)) (CAT$morphism ?c))
             (= (SET.FTN$composition (opsecond ?c) (CAT$source ?c))
                (SET.FTN$composition (cocone-diagram ?c) (object2 ?c)))
             (= (SET.FTN$composition (opsecond ?c) (CAT$target ?c))
                (opvertex ?c))))
```

- There is a function '`colimiting-cocone`' that maps a copair in a category *C* to its collection of coproduct cocones (this may be empty). The opvertex *C*-object of a colimiting cocone (Figure 2) is given by the function '`binary-coproduct`'. It comes equipped with two injection *C*-morphisms '`injection1`' and '`injection2`' given by the opfirst and opsecond of the colimiting cocone.

```
(10) (KIF$function colimiting-cocone)
     (KIF$signature colimiting-cocone CAT$category CNGfunction)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (colimiting-cocone ?c) (diagram ?c) SET$class)
              (forall (?r ((diagram ?c) ?r))
                  (SET$subclass
                      ((colimiting-cocone ?c) ?r)
                      ((SET.FTN$fiber (cocone-diagram ?c)) ?r)))))

(11) (KIF$function binary-coproduct)
     (KIF$signature binary-coproduct CAT$category CNG$function)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (binary-coproduct ?c) (diagram ?c) SET.FTN$function)
              (forall (?r ((diagram ?c) ?r))
                  (and (= (SET.FTN$source ((binary-coproduct ?c) ?r))
                          ((colimiting-cocone ?c) ?r))
                       (= (SET.FTN$target ((binary-coproduct ?c) ?r))
                          (CAT$object ?c))
                       (= ((binary-coproduct ?c) ?r)
                          (SET.FTN$composition
                              (SET.FTN$inclusion
                                  ((colimiting-cocone ?c) ?r) (cocone ?c))
                              (opvertex ?c)))))))

(12) (KIF$function injection1)
     (KIF$signature injection1 CAT$category CNG$function)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (injection1 ?c) (diagram ?c) SET.FTN$function)
              (forall (?r ((diagram ?c) ?r))
                  (and (= (SET.FTN$source ((injection1 ?c) ?r))
                          ((colimiting-cocone ?c) ?r))
                       (= (SET.FTN$target ((injection1 ?c) ?r))
                          (CAT$morphism ?c))
                       (= (SET.FTN$composition ((injection1 ?c) ?r) (CAT$source ?c))
```

```
                        (SET.FTN$composition
                            (SET.FTN$inclusion
                                ((colimiting-cocone ?c) ?r) (cocone ?c))
                            (SET.FTN$composition (cocone-diagram ?c) (object1 ?c))))
                    (= (SET.FTN$composition ((injection1 ?c) ?r) (CAT$target ?c))
                       ((binary-coproduct ?c) ?r))
                    (= ((injection1 ?c) ?r)
                       (SET.FTN$composition
                           (SET.FTN$inclusion
                               ((colimiting-cocone ?c) ?r) (cocone ?c))
                           (opfirst ?c)))))))

(13) (KIF$function injection2)
     (KIF$signature injection2 CAT$category CNG$function)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (injection2 ?c) (diagram ?c) SET.FTN$function)
              (forall (?r ((diagram ?c) ?r))
                  (and (= (SET.FTN$source ((injection2 ?c) ?r))
                          ((colimiting-cocone ?c) ?r))
                       (= (SET.FTN$target ((injection2 ?c) ?r))
                          (CAT$morphism ?c))
                       (= (SET.FTN$composition ((injection2 ?c) ?r) (CAT$source ?c))
                          (SET.FTN$composition
                              (SET.FTN$inclusion
                                  ((colimiting-cocone ?c) ?r) (cocone ?c))
                              (SET.FTN$composition (cocone-diagram ?c) (object2 ?c))))
                       (= (SET.FTN$composition ((injection2 ?c) ?r) (CAT$target ?c))
                          ((binary-coproduct ?c) ?r))
                       (= ((injection2 ?c) ?r)
                          (SET.FTN$composition
                              (SET.FTN$inclusion
                                  ((colimiting-cocone ?c) ?r) (cocone ?c))
                              (opsecond ?c)))))))
```

- For any coproduct diagram in a category *C* and any colimiting-cocone with that diagram as its base, there is a function that maps any cocone with the same base diagram to a unique *comediator C*-morphism whose source is the binary coproduct and whose target is opvertex of the cocone. This is the unique morphism that commutes with opfirst and opsecond morphisms. We use a KIF definite description abbreviation to define this. Existence and uniqueness represents the universality of the coproduct operator. A derived theorem states that all binary coproducts are isomorphic in *C*.

```
(14) (KIF$function comediator)
     (KIF$signature comediator CAT$category KIF$function)
     (forall (?c (CAT$category ?c))
         (and (KIF$signature (comediator ?c) (diagram ?c) CNG$function)
              (forall (?r ((diagram ?c) ?r))
                  (and (CNG$signature ((comediator ?c) ?r)
                           ((colimiting-cocone ?c) ?r) SET.FTN$function)
                       (forall (?s (((colimiting-cocone ?c) ?r) ?s))
                           (= (((comediator ?c) ?r) ?s)
                              (the (?f (SET.FTN$function ?f))
                                  (and (= (SET.FTN$source ?f)
                                          ((fiber (cocone-diagram ?c)) ?r))
                                       (= (SET.FTN$target ?f)
                                          (CAT$morphism ?c))
                                       (= (SET.FTN$composition ?f (CAT$source ?c))
                                          ((SET.TOP$diagonal
                                              ((fiber (cocone-diagram ?c)) ?r)
                                              (CAT$object ?c))
                                            (((binary-coproduct ?c) ?r) ?s)))
                                       (= (SET.FTN$composition ?f (CAT$target ?c))
                                          (SET.FTN$composition
                                              (SET.FTN$inclusion
                                                  ((fiber (cocone-diagram ?c)) ?r) (cocone ?c))
                                              (opvertex ?c)))))))))))
```

Coequalizers

COL.COEQU

. . .

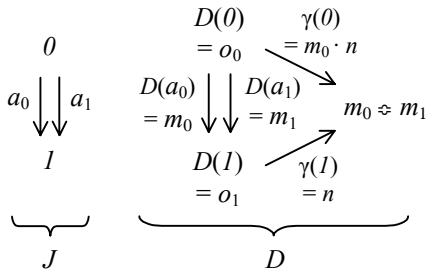


Figure 3: Coequalizer

Pushouts

COL.PSH

Given a category *C*, a *pushout* in *C* (Figure 4) is a finite colimit for a diagram of shape $J = \bullet \leftarrow \bullet \rightarrow \bullet$. Such a diagram (of *C*-objects and *C*-morphisms) is called an *span*.

- A *span* is the appropriate base diagram for a pushout. Each span consists of a pair of *C*-morphisms called *first* and *second*. These are required to have a common source *C*-object called the *vertex*. Let '`span`' be the COL.PSH namespace term that denotes the *Span* collection. This is synonymous with the phrase "pushout diagram". Spans are determined by their pair of component functions.

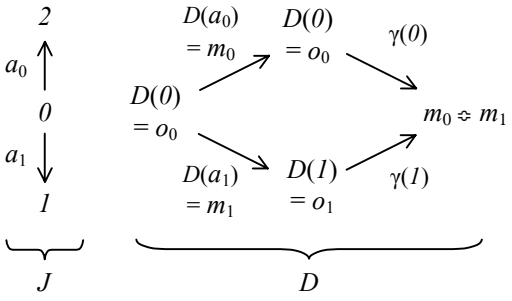


Figure 4: Pushout

```
(1) (CNG$function diagram)
    (CNG$function span)
    (= span diagram)
    (CNG$signature diagram CAT$category SET$class)

(2) (CNG$function object1)
    (CNG$signature object1 CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (object1 ?c)) (diagram ?c))
             (= (SET.FTN$target (object1 ?c)) (CAT$object ?c))))

(3) (CNG$function object2)
    (CNG$signature object2 CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (object2 ?c)) (diagram ?c))
             (= (SET.FTN$target (object2 ?c)) (CAT$object ?c))))

(4) (CNG$function vertex)
    (CNG$signature vertex CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (vertex ?c)) (diagram ?c))
             (= (SET.FTN$target (vertex ?c)) (CAT$object ?c))))

(5) (CNG$function first)
    (CNG$signature first CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (first ?c)) (diagram ?c))
             (= (SET.FTN$target (first ?c)) (CAT$morphism ?c))
             (= (SET.FTN$composition (first ?c) (CAT$source ?c))
                (vertex ?c))
             (= (SET.FTN$composition (first ?c) (CAT$target ?c))
                (object1 ?c))))
```

```
(6) (CNG$function second)
    (CNG$signature second CAT$category SET.FTNfunction)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (second ?c)) (diagram ?c))
             (= (SET.FTN$target (second ?c)) (CAT$morphism ?c))
             (= (SET.FTN$composition (second ?c) (CAT$source ?c))
                (vertex ?c))
             (= (SET.FTN$composition (second ?c) (CAT$target ?c))
                (object2 ?c))))

    (forall (?c (CAT$category ?c)
             ?r1 ((diagram ?c) ?r1) ?r2 ((diagram ?c) ?r2))
        (=> (and (= ((first ?c) ?r1) ((first ?c) ?r2))
                 (= ((second ?c) ?r1) ((second ?c) ?r2)))
            (= ?r1 ?r2)))
```

- The *copair* of target C-objects (postfixing discrete diagram) of any span (pushout diagram) in a category *C* is named.

```
(7) (CNG$function copair)
    (CNG$signature copair CAT$category SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (copair ?c)) (diagram ?c))
             (= (SET.FTN$target (copair ?c)) (COL.COPRD$diagram))))
             (= (SET.FTN$composition (copair ?c) (COL.COPRD$object1 ?c))
                (object1 ?c))
             (= (SET.FTN$composition (copair ?c) (COL.COPRD$object2 ?c))
                (object2 ?c))))
```

- Every span in *C* has an opposite.

```
(8) (CNG$function opposite)
    (CNG$signature opposite CAT$category SET.FTN$function)
    (forall (?c (CAT$category ?c))
        (and (= (SET.FTN$source (opposite ?c)) (diagram ?c))
             (= (SET.FTN$target (opposite ?c)) (diagram ?c))
             (= (SET.FTN$composition (opposite ?c) (object1 ?c))
                (object2 ?c))
             (= (SET.FTN$composition (opposite ?c) (object2 ?c))
                (object1 ?c))
             (= (SET.FTN$composition (opposite ?c) (vertex ?c))
                (vertex ?c))
             (= (SET.FTN$composition (opposite ?c) (first ?c))
                (second ?c))
             (= (SET.FTN$composition (opposite ?c) (second ?c))
                (first ?c))))
```

- The opposite of the opposite is the original span – the following theorem can be proven.

```
    (forall (?c (CAT$category ?c))
        (= (SET.FTN$composition (opposite ?c) (opposite ?c))
           (SET.FTN$identity (diagram ?c))))
```

- *Pushout cocones* are used to specify and axiomatize pushouts in a category *C*. Each pushout cocone has an underlying pushout *diagram*, an *opvertex C*-object, and a pair of *C*-morphisms called *opfirst* and *opsecond*, whose common target *C*-object is the opvertex and whose source *C*-objects are the target *C*-objects of the *C*-morphisms in the underlying diagram. The opfirst and opsecond morphisms form a commutative diagram with the span, implicitly making the pushout cocone a natural transformation. A pushout cocone is the very special case of a general colimit cocone over a pushout diagram. Let ‘cocone’ be the COL.PSH namespace term that denotes the *Pushout Cocone* collection.

```
(9) (CNG$function cocone)
    (CNG$signature cocone CAT$category SET$class)

(10) (CNG$function cocone-diagram)
     (CNG$signature cocone-diagram CAT$category SET.FTNfunction)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (cocone-diagram ?c)) (cocone ?c))
              (= (SET.FTN$target (cocone-diagram ?c)) (diagram ?c))))
```

```
(11) (CNG$function opvertex)
     (CNG$signature opvertex CAT$category SET.FTNfunction)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (opvertex ?c)) (cocone ?c))
              (= (SET.FTN$target (opvertex ?c)) (CAT$object ?c))))

(12) (CNG$function opfirst)
     (CNG$signature opfirst CAT$category SET.FTNfunction)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (opfirst ?c)) (cocone ?c))
              (= (SET.FTN$target (opfirst ?c)) (CAT$morphism ?c))
              (= (SET.FTN$composition (opfirst ?c) (CAT$source ?c))
                 (SET.FTN$composition (cocone-diagram ?c) (object1 ?c)))
              (= (SET.FTN$composition (opfirst ?c) (CAT$target ?c))
                 (opvertex ?c))))

(13) (CNG$function opsecond)
     (CNG$signature opsecond CAT$category SET.FTNfunction)
     (forall (?c (CAT$category ?c))
         (and (= (SET.FTN$source (opsecond ?c)) (cocone ?c))
              (= (SET.FTN$target (opsecond ?c)) (CAT$morphism ?c))
              (= (SET.FTN$composition (opsecond ?c) (CAT$source ?c))
                 (SET.FTN$composition (cocone-diagram ?c) (object2 ?c)))
              (= (SET.FTN$composition (opsecond ?c) (CAT$target ?c))
                 (opvertex ?c))))
```

- There is a function '`colimiting-cocone`' that maps a span in a category *C* to its collection of pushout cocones (this may be empty). The opvertex *C*-object of a colimiting cocone (Figure 4) is given by the function '`pushout`'. It comes equipped with two `injection` *C*-morphisms '`injection1`' and '`injection2`' given by the opfirst and opsecond of the colimiting cocone.

```
(14) (KIF$function colimiting-cocone)
     (KIF$signature colimiting-cocone CAT$category CNGfunction)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (colimiting-cocone ?c) (diagram ?c) SET$class)
              (forall (?r ((diagram ?c) ?r))
                  (SET$subclass
                      ((colimiting-cocone ?c) ?r)
                      ((SET.FTN$fiber (cocone-diagram ?c)) ?r)))))

(15) (KIF$function pushout)
     (KIF$signature pushout CAT$category CNG$function)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (pushout ?c) (diagram ?c) SET.FTN$function)
              (forall (?r ((diagram ?c) ?r))
                  (and (= (SET.FTN$source ((pushout ?c) ?r))
                          ((colimiting-cocone ?c) ?r))
                       (= (SET.FTN$target ((pushout ?c) ?r))
                          (CAT$object ?c))
                       (= ((pushout ?c) ?r)
                          (SET.FTN$composition
                              (SET.FTN$inclusion
                                  ((colimiting-cocone ?c) ?r) (cocone ?c))
                              (opvertex ?c)))))))

(16) (KIF$function injection1)
     (KIF$signature injection1 CAT$category CNG$function)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (injection1 ?c) (diagram ?c) SET.FTN$function)
              (forall (?r ((diagram ?c) ?r))
                  (and (= (SET.FTN$source ((injection1 ?c) ?r))
                          ((colimiting-cocone ?c) ?r))
                       (= (SET.FTN$target ((injection1 ?c) ?r))
                          (CAT$morphism ?c))
                       (= (SET.FTN$composition ((injection1 ?c) ?r) (CAT$source ?c))
                          (SET.FTN$composition
                              (SET.FTN$inclusion
                                  ((colimiting-cocone ?c) ?r) (cocone ?c))
                              (SET.FTN$composition (cocone-diagram ?c) (object1 ?c))))
```

```
                        (= (SET.FTN$composition ((injection1 ?c) ?r) (CAT$target ?c))
                           ((pushout ?c) ?r))
                        (= ((injection1 ?c) ?r)
                           (SET.FTN$composition
                               (SET.FTN$inclusion
                                   ((colimiting-cocone ?c) ?r) (cocone ?c))
                               (opfirst ?c)))))))

(17) (KIF$function injection2)
     (KIF$signature injection2 CAT$category CNG$function)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (injection2 ?c) (diagram ?c) SET.FTN$function)
              (forall (?r ((diagram ?c) ?r))
                  (and (= (SET.FTN$source ((injection2 ?c) ?r))
                          ((colimiting-cocone ?c) ?r))
                       (= (SET.FTN$target ((injection2 ?c) ?r))
                          (CAT$morphism ?c))
                       (= (SET.FTN$composition ((injection2 ?c) ?r) (CAT$source ?c))
                          (SET.FTN$composition
                              (SET.FTN$inclusion
                                  ((colimiting-cocone ?c) ?r) (cocone ?c))
                              (SET.FTN$composition (cocone-diagram ?c) (object2 ?c))))
                       (= (SET.FTN$composition ((injection2 ?c) ?r) (CAT$target ?c))
                          ((pushout ?c) ?r))
                       (= ((injection2 ?c) ?r)
                          (SET.FTN$composition
                              (SET.FTN$inclusion
                                  ((colimiting-cocone ?c) ?r) (cocone ?c))
                              (opsecond ?c)))))))
```

- For any pushout diagram in a category *C* and any colimiting-cocone with that diagram as its base, there is a function that maps any cocone with the same base diagram to a unique *comediator C*-morphism whose source is the pushout and whose target is opvertex of the cocone. This is the unique morphism that commutes with opfirst and opsecond morphisms. We use a KIF definite description abbreviation to define this. Existence and uniqueness represents the universality of the pullback operator. A derived theorem states that all pushouts are isomorphic in *C*.

```
(18) (KIF$function comediator)
     (KIF$signature comediator CAT$category KIF$function)
     (forall (?c (CAT$category ?c))
         (and (KIF$signature (comediator ?c) (diagram ?c) CNG$function)
              (forall (?r ((diagram ?c) ?r))
                  (and (CNG$signature ((comediator ?c) ?r)
                           ((colimiting-cocone ?c) ?r) SET.FTN$function)
                       (forall (?s (((colimiting-cocone ?c) ?r) ?s))
                           (= (((comediator ?c) ?r) ?s)
                              (the (?f (SET.FTN$function ?f))
                                  (and (= (SET.FTN$source ?f)
                                          ((fiber (cocone-diagram ?c)) ?r))
                                       (= (SET.FTN$target ?f)
                                          (CAT$morphism ?c))
                                       (= (SET.FTN$composition ?f (CAT$source ?c))
                                          ((SET.TOP$diagonal
                                              ((fiber (cocone-diagram ?c)) ?r)
                                              (CAT$object ?c))
                                            (((pushout ?c) ?r) ?s)))
                                       (= (SET.FTN$composition ?f (CAT$target ?c))
                                          (SET.FTN$composition
                                              (SET.FTN$inclusion
                                                  ((fiber (cocone-diagram ?c)) ?r) (cocone ?c))
                                              (opvertex ?c)))))))))))
```

General Colimits

- Given a category C, which serves as an environment within which colimits are to be constructed, a *diagram* D in C is a functor from some shape or indexing category J to the base category C. In the KIF formalism below the *shape* of a diagram in C is defined in terms of the '`FUNC$source`' predicate.

```
(1) (CNG$function diagram)
    (CNG$signature diagram CAT$category CNG$conglomeration)
    (forall (?c (CAT$category ?c))
        (and (CNG$subconglomeration (diagram ?c) FUNC$functor)
             (forall (?d (FUNC$functor ?d))
                 (<=> ((diagram ?c) ?d)
                      (= (FUNC$target ?d) ?c)))))

(2) (KIF$function shape)
    (KIF$signature shape CAT$category CNG$function)
    (forall (?c (CAT$category ?c))
        (and (CNG$signature (shape ?c) (diagram ?c) CAT$category)
             (forall (?d ((diagram ?c) ?d))
                 (= ((shape ?c) ?d) (FUNC$source ?d)))))
```

- Given a category C a *cocone* $\tau : D \Rightarrow \Delta_{J,C}(o) : J \to C$ (Figure 5) is a natural transformation from a diagram D in the category C of some shape $J = src(D)$ to the constant functor $\Delta_{J,C}(o)$ for some object $o \in obj(C)$. The source functor D is called the *base* of the cocone τ. The object $o \in obj(C)$ is called the *vertex* of the cocone τ. A cocone is analogous to the upper bound of a subset of a partial order – the order is analogous to the category C, the chosen subset is analogous to the base diagram D, and the upper bound element is analogous to the vertex C-object o. We use a KIF definite description to define the vertex. The vertex function restricts cocones to be natural transformations whose target is a constant function.

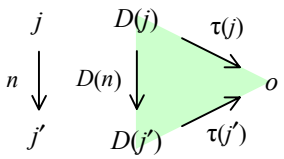


Figure 5: Cocone

```
(3) (KIF$function cocone)
    (KIF$signature cocone CAT$category CNG$conglomerate)
    (forall (?c (CAT$category ?c))
        (and (CNG$subconglomerate (cocone ?c) NAT$natural-transformation)
             (forall (?tau ((cocone ?c) ?tau))
                 (= (NAT$target-category ?tau) ?c))))

(4)  (KIF$function cocone-diagram)
     (KIF$function base)
     (= base cocone-diagram)
     (KIF$signature cocone-diagram CAT$category CNG$function)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (cocone-diagram ?c) (cocone ?c) (diagram ?c))
              (forall (?tau ((cocone ?c) ?tau))
                  (= ((cocone-diagram ?c) ?tau)
                     (NAT$source-functor ?tau)))))

(5)  (KIF$function opvertex)
     (KIF$signature opvertex CAT$category CNG$function)
     (forall (?c (CAT$category ?c))
         (and (CNG$signature (opvertex ?c) (cocone ?c) (CAT$object ?c))
              (forall (?tau ((cocone ?c) ?tau))
                  (= ((opvertex ?c) ?tau)
                     (the (?o ((CAT$object ?c) ?o))
                         (= (NAT$target-functor ?tau)
                            ((FUNC$diagonal (NAT$source-category ?tau) ?c) ?o)))))))
```

- Colimits are the vertices of special cocones. Given a category C a *colimiting-cocone* in C (Figure 6) is a universal cocone – it consists of a cocone from a diagram D in C of some shape $J = src(D)$ to a `op-vertex` $\bar{o} \in obj(C)$

  $\gamma : D \Rightarrow \Delta_{C,J}(\bar{o}) : J \to C$

  that is *universal*: for any other cocone

  $\tau : D \Rightarrow \Delta_{C,J}(o) : J \to C$

with the same base diagram, there is a unique morphism $m : \bar{o} \to o$ with $\gamma(j) \cdot m = \tau(j)$ for any indexing object $j \in Obj(J)$, or equivalently as natural transformations, with $\gamma \bullet \Delta_{J,C}(m) = \tau$. The collection of colimiting cocones may be empty – it is empty if no colimits for the diagram D exist in the category C. If it is nonempty for any diagram of shape J, then C is said to have *J-colimits*. A category C is *cocomplete*, if it has J-colimits for any shape J. The `opvertex` $\bar{o}$ of the colimiting-cocone γ is called a *colimit*. Let $colim_C(D)$ denote the function that takes a colimiting cocone to its `opvertex`, an object of C. This will be the empty function, if no colimits exist for diagram D. In the same way that a cocone is analogous to the upper bound, a colimit is analogous to a least upper bound (or supremum) of a subset of a partial order.

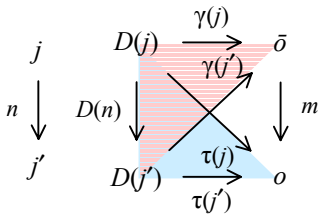


Figure 6: Colimiting Cocone

Axioms (6–7) formalize colimits. Axiom (6) defines a unary KIF function '`(colimiting-cocone ?c)`' that maps a diagram to its colimit conglomerate. The `opvertex` of a colimiting cocone is a particular colimit represented in axiom (7) by the CNG function '`((colimiting-cocone ?c) ?d)`'.

```
(6) (KIF$function colimiting-cocone)
    (KIF$signature colimiting-cocone CAT$category KIF$function)
    (forall (?c (CAT$category ?c))
        (and (KIF$signature (colimiting-cocone ?c) (diagram ?c) CNG$conglomerate)
             (forall (?d ((diagram ?c) ?d))
                 (and (CNG$subconglomerate
                          ((colimiting-cocone ?c) ?d)
                          (cocone ?c))
                      (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                          (=> (((colimiting-cocone ?c) ?d) ?g)
                              (= ?d ((cocone-diagram ?c) ?g)))))))))

(7)  (KIF$function colimit)
     (KIF$signature colimit CAT$category KIF$function)
     (forall (?c (CAT$category ?c))
         (and (KIF$signature (colimit ?c) (diagram ?c) CNG$function)
              (forall (?d ((diagram ?c) ?d))
                  (and (CNG$signature ((colimit ?c) ?d)
                           ((colimiting-cocone ?c) ?d) (CAT$object ?c))
                       (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                           (= (((colimit ?c) ?d) ?g)
                              ((opvertex ?c) ?g)))))))
```

For any diagram D and any colimiting-cocone γ with base D, let $m = comediator_{C,D}(\gamma)$ (Figure 6) denote the function that takes any cocone τ of base D and returns its unique comediating C-morphism whose source is the colimit and whose target is the opvertex of the cocone.

Axiom (8) formalizes comediators. There is a comediator function '`(((comediator ?c) ?d) ?g)`' from the colimit of the diagram to the opvertex of a cocone over a diagram. This is the unique function that commutes with colimiting cocone and given cocone. We use a KIF definite description abbreviation to define this. Existence and uniqueness represents the universality of the colimit.

```
(8) (KIF$function comediator)
    (KIF$signature colimit (CAT$category KIF$function)
    (forall (?c (CAT$Category ?c))
      (and (KIF$signature (comediator ?c) (diagram ?c) KIF$function)))
           (forall (?d ((diagram ?c) ?d))
             (and (KIF$signature ((comediator ?c) ?d)
                    ((colimiting-cocone ?c) ?d) KIF$function)
                  (forall (?g (((colimiting-cocone ?c) ?d) ?g))
                    (and (KIF$signature (((comediator ?c) ?d) ?g)
                           (cocone ?c) (CAT$morphism ?c))
                         (forall (?t ((cocone ?c) ?t))
                           (= ((((comediator ?c) ?d) ?g) ?t)
                              (the (?m (CAT$morphism ?c) ?m))
                                (and (= ((CAT$source ?c) ?m) (((colimit ?c) ?d) ?g))
                                     (= ((CAT$target ?c) ?m) ((opvertex ?c) ?t))
                                     (= (NAT$vertical-composition
                                          ?g ((NAT$diagonal ((shape ?c) ?d) ?c) ?m))
                                        ?t)))))))))))
```

- If the colimiting cocone conglomerate is nonempty for any diagram of shape *J*, then C is said to have *J-colimits* and to be *J-cocomplete*. A category *C* is *small-cocomplete*, if it has colimits for any small diagram of *C*; that is, when it is *J*-cocomplete for all small shape categories J. Axiom (9) formalizes cocompleteness in terms of the colimiting cocone conglomerate '`((colimiting-cocone ?c) ?d)`'.

```
(9) (KIF$function cocomplete)
    (KIF$signature cocomplete CAT$category CNG$conglomerate)
    (forall (?j (CAT$category ?j))
        (and (CNG$subconglomerate (cocomplete ?j) CAT$category)
             (forall (?c (CAT$category ?c))
                 (<=> ((cocomplete ?j) ?c)
                      (forall (?d ((diagram ?c) ?d))
                          (=> (= ((shape ?c) ?d) ?j)
                              (exists (?g (((colimiting-cocone ?c) ?d) ?g)))))))))

(10) (CNG$conglomerate small-cocomplete)
     (CNG$subconglomerate small-cocomplete CAT$category)
     (forall (?c (CAT$category ?c))
         (<=> (small-cocomplete ?c)
              (forall (?j (CAT$category ?j))
                  ((cocomplete ?j) ?c))))
```

- It is a standard theorem that given a category *C* and a diagram *D* in the category *C*, any two colimits are isomorphic. The following KIF expresses this in an external namespace.

```
(forall (?c (CAT$category ?c) ?d ((diagram ?c) ?d)
        ?g1 (((colimiting-cocone ?c) ?d) ?g1)
        ?g2 (((colimiting-cocone ?c) ?d) ?g2))
    ((CAT$isomorphic ?c) (((colimit ?c) ?d) ?g1) (((colimit ?c) ?d) ?g2)))
```

Examples

The general KIF formulation for colimits can be related to several special kinds of finite colimits: initial objects, binary coproducts, coequalizers and pushouts.

- Suppose an initial object 0_C exists in a category *C*. Here we are interested in the initial (empty) shape category $J = \mathbf{0}$. If *D* is the empty diagram in the category *C* of shape $\mathbf{0}$, then the colimit object is the initial object 0_C and the colimiting cocone is the empty natural transformation (vertical identity at *D*) with target category *C*. Also, the mediator function to any object (cocone opvertex) is the *counique* function. (The initial object in Set is the empty set. The initial object in Classification is the classification with one instance, but no types.)

- Suppose binary coproducts exist in a category *C*. Consider the binary coproduct of any two objects o_0, $o_1 \in obj(C)$. Here we are interested in the shape category $J = set2$ of two discrete nodes *0* and *1*. If *D* is the diagram in the category *C* of shape *set2*, whose object function maps *0* and *1* to the *C*-objects o_0 and o_1, respectively, then the colimit object is the binary coproduct $o_0 + o_1$ and the colimiting cocone has as components the coproduct injections morphisms $i_0 : o_0 \to o_0+o_1$ and $i_1 : o_1 \to o_0+o_1$. This statement is represented as the following KIF theorem.

```
(forall (?c (CAT$category ?c))
    (=> ((cocomplete set2) ?c)
      (forall ?d ((diagram ?c) ?d))
          (=> (= ((shape ?c) ?d) set2)
              (exists (?p (COL.COPRD$diagram ?p)
                       ?g (((colimiting-cocone ?c) ?d) ?g))
                  (and (= ((FUNC$object ?d) set2#0)
                          ((COL.COPRD$object1 ?c) ?p))
                       (= ((FUNC$object ?d) set2#1)
                          ((COL.COPRD$object2 ?c) ?p))
                       (= (((colimit ?c) ?d) ?g)
                          ((COL.COPRD$binary-coproduct ?c) ?p))
                       (= ((NAT$component ?g) set2#0)
                          ((COL.COPRD$injection1 ?c) ?p))
                       (= ((NAT$component ?g) set2#0)
                          ((COL.COPRD$injection2 ?c) ?p)))))))))
```

- Suppose coequalizers exist in a category C. Consider the coequalizer of any two parallel C-morphisms $m_0, m_1 : o_0 \to o_1$. Here we are interested in the shape category $J = parpair$ of two parallel edges. If D is the diagram in the category C of shape *parpair*, whose object function maps *0* and *1* to the C-objects o_0 and o_1, respectively, and whose morphism function maps a_0 and a_1 to the C-morphisms m_0 and m_1, respectively, then the colimit object is the coequalizer $m_0 \div m_1$ and the colimiting cocone has as its *1*-component the canonical morphism $n : o_1 \to m_0 \div m_1$ and has as *0*-component is the C-composition $m_0 \cdot n = m_1 \cdot n : o_0 \to m_0 \div m_1$. This statement is represented as the following KIF theorem.

```
(forall (?c (CAT$category ?c))
    (=> ((cocomplete parpair) ?c)
        (forall ?d ((diagram ?c) ?d))
            (=> (= ((shape ?c) ?d) parpair)
                (exists (?pp (COL.COEQU$diagram ?pp)
                         ?g (((colimiting-cocone ?c) ?d) ?g))
                    (and (= ((FUNC$object ?d) parpair#0)
                            ((COL.COEQU$object1 ?c) ?pp))
                         (= ((FUNC$object ?d) parpair#1)
                            ((COL.COEQU$object2 ?c) ?pp))
                         (= ((FUNC$morphism ?d) parpair#a0)
                            ((COL.COEQU$morphism1 ?c) ?pp))
                         (= ((FUNC$morphism ?d) parpair#a1)
                            ((COL.COEQU$morphism2 ?c) ?pp))
                         (= (((colimit ?c) ?d) ?g)
                            ((COL.COEQU$coequalizer ?c) ?pp))
                         (= ((NAT$component ?g) parpair#0)
                            ((CAT$composition ?c)
                                [((COL.COEQU$morphism1 ?c) ?pp)
                                 ((COL.COEQU$canon ?c) ?pp)]))
                         (= ((NAT$component ?g) set2#0)
                            ((COL.COEQU$canon ?c) ?pp))))))))
```

- Suppose pushouts exist in a category C. Consider the pushout of any span of C-morphisms $m_1 : o_0 \to o_1$ and $m_2 : o_0 \to o_2$. Here we are interested in the shape category $J = span$. If D is the diagram in the category C of shape *span*, whose object function maps *0, 1* and *2* to the C-objects o_0, o_1 and o_2, respectively, and whose morphism function maps a_1 and a_2 to the C-morphisms m_1 and m_2, respectively, then the colimit object is the pushout $o_1 +_o o_2$ and the colimiting cocone has as components the pushout injection morphisms $i_1 : o_1 \to o_1 \times_o o_2$ and $i_2 : o_2 \to o_1 \times_o o_2$. The analogous theorem can be proven.

The Namespace of Large Kan Extensions

The Namespace of Large Classifications

IF Theories

IF Logics

The Namespace of Large Concept Lattices

The Namespace of Large Topoi

Part II: The Small Aspect

The Model Theory Ontology

The Namespace of Small Sets

This is mostly finished.

The Namespace of Small Relations

The Namespace of Small Classifications

This is all finished.

The Namespace of Small Spans and Hypergraphs

A hypergraph is equivalent to a span. The span encoding is finished. The hypergraph equivalent is finished in theory, but has not yet been coded.

The Namespace of Structures (Models)

A model is a two-dimensional structure with a classification along the instance-type distinction and a hypergraph along the entity-relation distinction The theory is finished, but the code has not been started.

References

Adámek, Jirí, Herrlich, Horst and Strecker, George E. 1990. ***Abstract and Concrete Categories***. New York: Wiley and Sons.

Barr, Michael. 1996. The Chu Construction. *Theory and Applications of Categories* 2.

Barwise, Jon and Seligman, Jerry. 1997. ***Information Flow: The Logic of Distributed Systems***. Cambridge University Press.

Ganter, Bernhard and Wille, Rudolf. 1999. ***Formal Concept Analysis: Mathematical Foundations***. Berlin/Heidelberg: Springer-Verlag.

Kent, Robert E. 2000. The Information Flow Foundation for Conceptual Knowledge Organization. *Proceedings of the 6th International Conference of the International Society for Knowledge Organization (ISKO)*. Toronto, Canada.

Mac Lane, Saunders. 1971. ***Categories for the Working Mathematician***, New York/Heidelberg/Berlin: Springer-Verlag. New edition (1998).