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

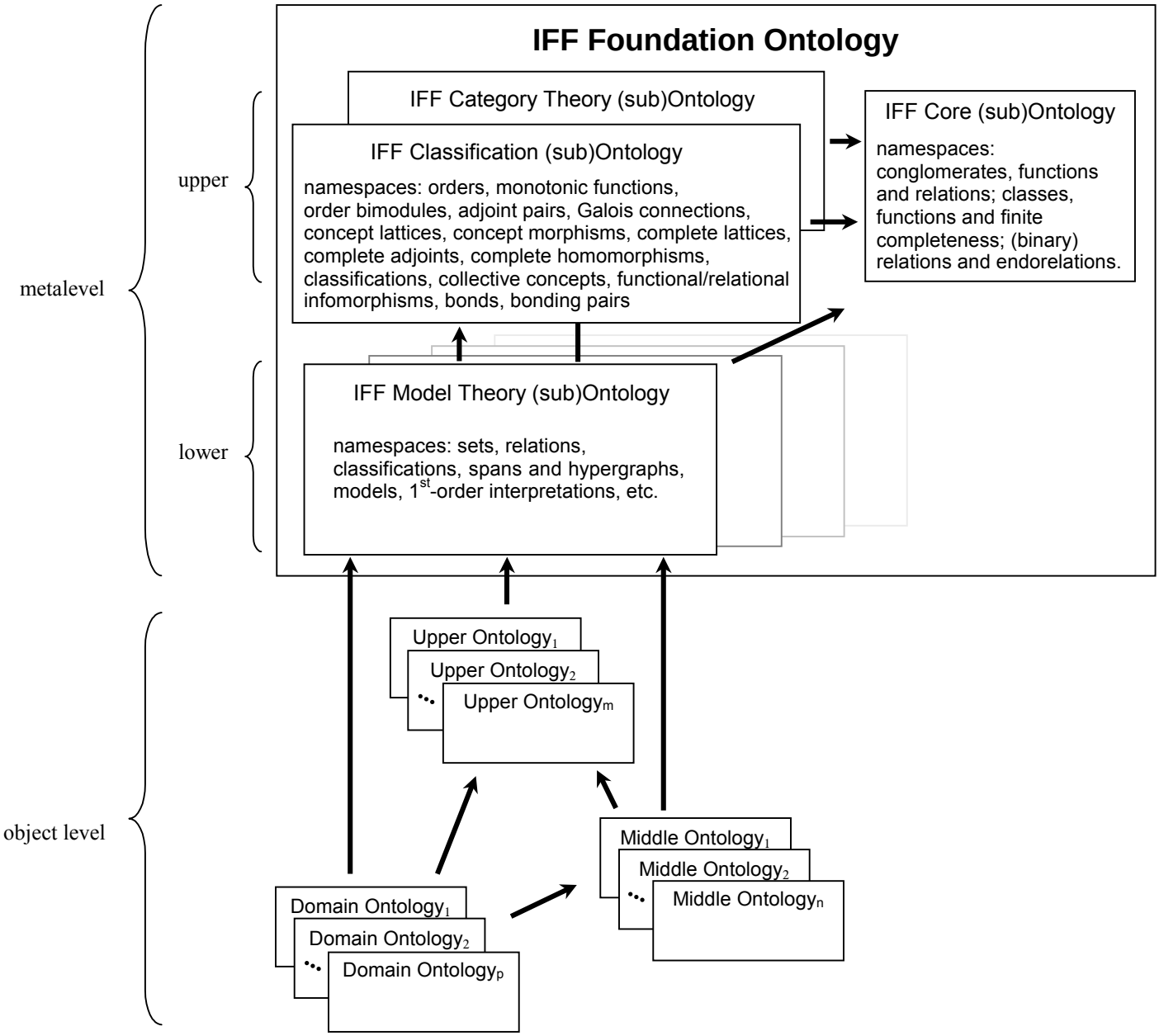


Figure 1: IFF Foundation Ontology Architecture (with dependencies)

[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 $\boldsymbol{L}$ is the meta-classification, whose instances are $\boldsymbol{L}$-structures (models), whose types are $\boldsymbol{L}$-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.

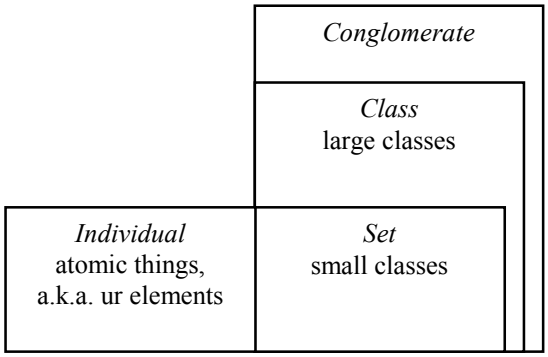


Figure 2: Collection Hierarchy

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`.

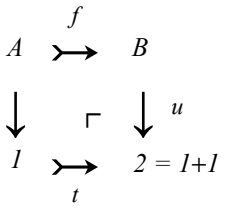


Diagram 1: Subobject Classifier

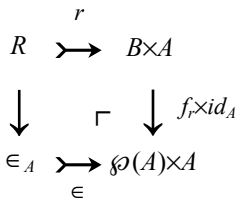


Diagram 2: Power Objects

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 epimorphism. 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.

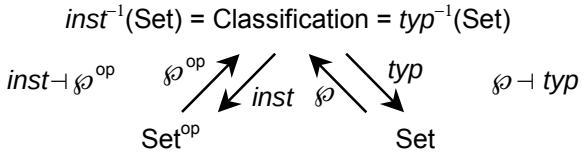


Figure 3: The Classification Fibered Span

Table 2 lists the correspondence between standard mathematical notation and the ontological terminology in the namespace for logic.

Table 2: Correspondence between Mathematical Notation and Ontological Terminology

Math	Ontological Terminology	Natural Language Description
$\forall$	`forall`	universal quantifier
$\exists$	`exists`	existential quantifier
$\wedge$	`and`	conjunction
$\vee$	`or`	disjunction
$\neg$	`not`	negation
$\rightarrow$	`=>`	implication
$\leftrightarrow$	`<=>`	equivalence

Table 3 lists the correspondence between standard mathematical notation and the ontological terminology in the basic ontology.

Table 3: Correspondence between Mathematical Notation and Ontological Terminology

Math	Ontological Terminology	Natural Language Description
A	'`KIF$class`'	KIF class – corresponds to a set-theory collection
R	'`KIF$relation`'	relation
f	'`KIF$function`'	KIF function – a functional relation
$R \subseteq A_1 \times \ldots \times A_n$	'`KIF$signature`'	the signature for a relation
$(a_1, \ldots, a_n)$	'`[?a1 ... ?an]`'	sequence notation

Table 4 (needs much expansion) lists the correspondence between standard mathematical notation and the ontological terminology in the namespace for classes, functions, and finite limits.

Table 4: Correspondence between Mathematical Notation and Ontological Terminology

Math	Ontological Terminology	Natural Language Description
$\subseteq$	'`SET$subclass`'	the subclass inclusion relation
$\cong$		the isomorphism relation between objects
$\varnothing$	'`SET.LIM$null`', '`SET.LIM$initial`'	the empty class – this is the initial object in the quasi-category of classes and functions
$\times$	'`SET.LIM.PRD$binary-product`'	binary product operator on objects

Part I: The Large Aspect

The Core Ontology

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.

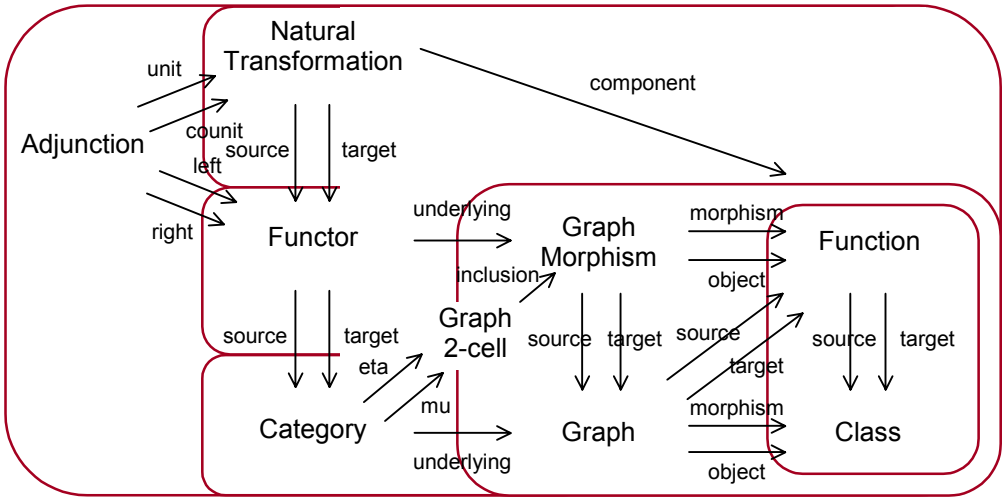


Diagram 1: Core Conglomerates and Functions

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 (KIF$length [@cng])))
                      (conglomerate ([@cng] ?n))))))
```

  - The binary Cartesian product of two conglomerates is defined in axiom (6).

```
(6) (KIF$function binary-product)
    (signature binary-product conglomerate conglomerate conglomerate)
    (forall (?c1 (conglomerate ?c1) ?c2 (conglomerate ?c2) ?z)
        (<=> ((binary-product ?c1 ?c2) ?z)
             (and (KIF$pair ?z)
                  (?c1 (?z 1))
                  (?c2 (?z 2)))))
```

  - 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.

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

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

(9) (KIF$function conglomerate2)
    (KIF$signature conglomerate2 relation conglomerate)
    (forall (?r (relation ?r))
        (KIF$signature ?r (conglomerate1 ?r) (conglomerate2 ?r)))

(10) (KIF$function extent)
     (KIF$signature extent relation conglomerate)
     (forall (?r (relation ?r))
         (and (subconglomerate
                  (extent ?r)
                  (binary-product (conglomerate1 ?r) (conglomerate2 ?r)))
```

```
                (forall (?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.

```
(11) (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'	'power'	'binary-union' 'binary-intersection'
SET .FTN	'function' 'parallel-pair' 'injection', 'surjection', 'bijection' 'monomorphism', 'epimorphism', 'isomorphism'	'source', 'target', 'identity' 'image', 'inclusion', 'fiber', 'inverse-image', 'power', 'direct-image' 'singleton', 'union', 'intersection'	'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 *⊆ conglomerate × conglomerate* *signature ⊆ function × KIF$sequence* *subclass ⊆ class × class* *disjoint ⊆ class × class* *partition ⊆ class × KIF$sequence* *restriction ⊆ function × CNG$function* *restriction-* *pullback ⊆ function × CNG$function*	*source, target* : *function → class* *identity, range* : *function → class* *vertex* : *span → class* *first*, *second* : *span → function* *opvertex* : *opspan → class* *opfirst*, *opsecond* : *opspan → function* *opposite* : *opspan → opspan*	*composition* : *function × function → function*
	unique : *class → function* *cone-opspan* : *cone → opspan* *vertex* : *cone → class* *first*, *second*, *mediator* : *cone → function* *limiting-cone* : *opspan → cone*	*binary-product* : *class × class → class* *binary-product-* *opspan* : *class × class → opspan*
	power : *class → relation*	*exponent* : *class × class → class* *evaluation* : *class × class → 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 on classes.

```
(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 pair of classes there is a binary union class and a binary intersection class.

```
(5) (CNG$function binary-union)
    (forall (?c1 (class ?c1) ?c2 (class ?c2) ?x)
        (<=> ((binary-union ?c1 ?c2) ?x)
             (or (?c1 ?x) (?c2 ?x))))

(6) (CNG$function binary-intersection)
    (forall (?c1 (class ?c1) ?c2 (class ?c2) ?x)
        (<=> ((binary-intersection ?c1 ?c2) ?x)
             (and (?c1 ?x) (?c2 ?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.

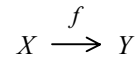


Figure 1: Class Function

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.

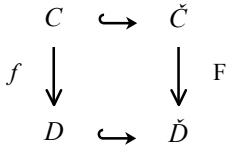


Diagram 1: Ordinary restriction

```
(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))))))
```

- 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)))))))
```

- Following the assumption that the power of a class is a class, we also assume that the power of a class function is a class function. For any class function $f : A \to B$ there is a *power* or *direct image* function $\wp f : \wp A \to \wp B$ defined by $\wp f(X) = \{y \in B \mid y = f(x) \text{ some } x \in X\} \subseteq B$ for any subset $X \subseteq A$.

```
(24) (CNG$function power)
     (CNG$function direct-image)
     (= power direct-image)
     (CNG$signature power function function)
     (forall (?f (function ?f))
         (and (= (source (power ?f)) (SET$power (source ?f)))
              (= (target (power ?f)) (SET$power (target ?f)))))
     (forall (?f (function ?f)
              ?X ((SET$power (source ?f)) ?X)
              ?y ((target ?f) ?y))
         (<=> (((power ?f) ?X) ?y)
              (exists (?x (?X ?x)) (= ?y (?f ?x)))))
```

- Clearly the image is related to the power as follows.

```
     (forall (?f (function ?f))
         (= (image ?f)
            ((power ?f) (source ?f))))
```

- For any class C there is a singleton function $\{\text{-}\}_C : C \to \wp C$ that embeds elements as subsets.

```
(25) (CNG$function singleton)
     (CNG$signature singleton SET$class SET.FTN$function)
     (forall (?c (SET$class ?c))
         (and (= (source (singleton ?c)) ?c)
              (= (target (singleton ?c)) (SET$power ?c))
              (forall (?x (?c ?x) ?y (?c ?y))
                  (<=> (((singleton ?c) ?x) ?y)
                       (= ?y ?x)))))
```

- In the presence of a preorder $\boldsymbol{A} = \langle A, \leq_A \rangle$, there are two ways that functions are transformed into binary relations – both by composition. For any function $f : B \to A$, the *left* relation $f_@ : A \to B$ is defined as

$$f_@(a, b) \text{ iff } a \leq_A f(b),$$

and the *right* relation $f^@ : B \to A$ as follows

$$f^@(b, a) \text{ iff } f(b) \leq_A a.$$

```
(26) (KIF$function left)
     (KIF$signature left preorder CNG$function)
     (forall (?o (preorder ?o))
         (CNG$signature (left ?o) function relation))
     (forall (?o (preorder ?o)
              ?f (function ?f))
         (<=> (exists (?r (relation ?r)) (= ((left ?o) ?f) ?r))
              (= (target ?f) (ORD$class ?o))))
     (forall (?o (preorder ?o) ?f (function ?f))
         (=> (= (target ?f) (ORD$class ?o))
             (and (= (source ((left ?o) ?f)) (ORD$class ?o))
                  (= (target ((left ?o) ?f)) (source ?f))
                  (forall (?a ((ORD$class ?o) ?a)
                           ?b ((source ?f) ?b))
                      (<=> ((extent ((left ?o) ?f)) [?a ?b])
                           ((extent ?o) [?a (?f ?b)])))))))

(27) (KIF$function right)
     (KIF$signature right preorder CNG$function)
     (forall (?o (preorder ?o))
         (CNG$signature (right ?o) function relation))
     (forall (?o (preorder ?o)
              ?f (function ?f))
         (<=> (exists (?r (relation ?r)) (= ((right ?o) ?f) ?r))
              (= (target ?f) (ORD$class ?o))))
     (forall (?o (preorder ?o) ?f (function ?f))
         (=> (= (target ?f) (ORD$class ?o))
```

```
            (and (= (source ((right ?o) ?f)) (source ?f))
                 (= (target ((right ?o) ?f)) (ORD$class ?o))
                 (forall (?b ((source ?f) ?b)
                             ?a ((ORD$class ?o) ?a))
                     (<=> ((extent ((right ?o) ?f)) [?b ?a])
                          ((extent ?o) [(?f ?b) ?a])))))
```

- Clearly, the function-to-relation function '`fn2rel`' can be expressed in terms of the right operator and the identity relation. It also can be expressed in terms of the opposite of the left operator.

```
(forall (?f (function ?f))
    (= (fn2rel ?f)
       ((right (ORD$identity (target ?f))) ?f)))

(forall (?f (function ?f))
    (= (fn2rel ?f)
       (REL$opposite ((left (ORD$identity (target ?f))) ?f))))
```

- For any class C, there is a union operator $\cup_C : \wp\wp C \to \wp C$ and an intersection operator $\cap_C : \wp\wp C \to \wp C$. That is, for any collection of subclasses $S \subseteq \wp C$ of a class C there is a union class $\cup_C(S)$ and an intersection class $\cap_C(S)$.

```
(28) (CNG$function union)
     (CNG$signature union SET$class SET.FTN$function)
     (forall (?c (SET$class ?c))
         (and (= (source (union ?c)) (SET$power ((SET$power ?c)))
              (= (target (union ?c)) (SET$power ?c))
              (forall (?S (SET$subclass S (SET$power ?c)) ?x (?c ?x))
                  (<=> (((union ?c) ?S) ?x)
                       (exists (?X (?S ?X)) (?X ?x))))))

(29) (CNG$function intersection)
     (CNG$signature intersection SET$class SET.FTN$function)
     (forall (?c (SET$class ?c))
         (and (= (source (intersection ?c)) (SET$power ((SET$power ?c)))
              (= (target (intersection ?c)) (SET$power ?c))
              (forall (?S (SET$subclass S (SET$power ?c)) ?x (?c ?x))
                  (<=> (((intersection ?c) ?S) ?x)
                       (forall (?X (?S ?X)) (?X ?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 products, equalizers, subequalizers and pullbacks are put into sub-namespaces. 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)))))
```

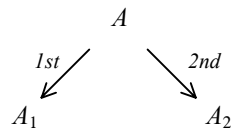


Figure 2: Product Cone

- 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)))))

(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))))
```

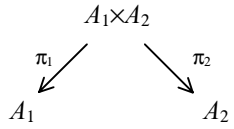


Figure 3: Limiting Cone

- 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)))))
```

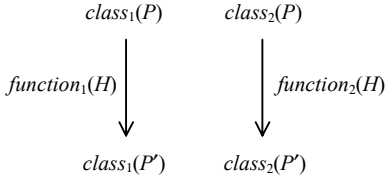


Figure 4: Function Pair

```
(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))))))
```

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)))))
```

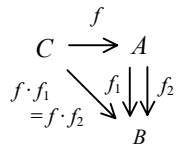


Figure 5: Equalizer Cone

- 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))
        (= (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)))
```

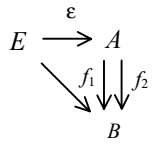


Figure 6: Limiting Cone

- 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))))
```

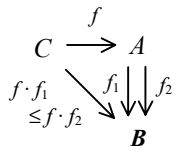


Figure 7: Subequalizer Cone

- 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))

(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))))
```

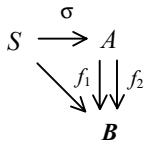


Figure 8: Lax Limiting Cone

```
(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 $\bullet \to \bullet \leftarrow \bullet$. 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.

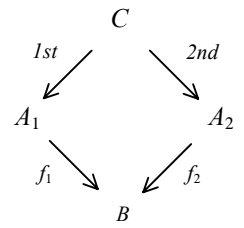


Figure 9: Pullback Cone

```
(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.

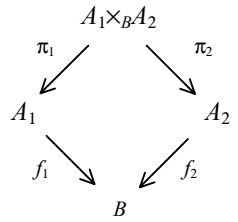


Figure 10: Limiting Cone

```
(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))))
```

- 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))))))
```

o 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

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

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

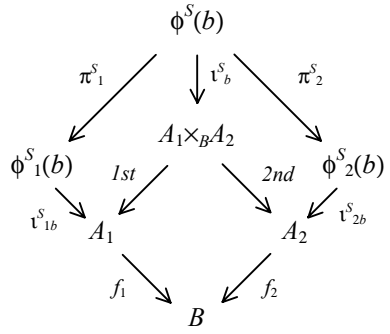


Diagram 2: Pullback Fibers

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 $(-)^{-1}$, we can define these as follows.

$$\phi^S = (I^{st} \cdot f_1)^{-1}$$

$$\phi^S{}_1 = f_1^{-1} \qquad \phi^S{}_{12} = f_1 \cdot f_2^{-1}$$

$$\phi^S{}_2 = f_2^{-1} \qquad \phi^S{}_{21} = f_2 \cdot f_1^{-1}$$

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)
```

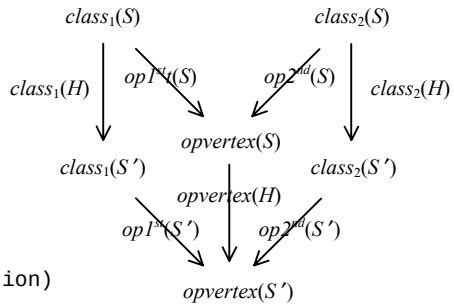


Figure 11: Opspan Morphism

```
(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)
    ((exponent natural-numbers natural-numbers) successor)

    (forall (?c (SET$class ?c)
             ?x ((element ?c) ?x)
             ?f ((exponent ?c ?c) ?f))
        (exists-unique (?h (SET.FTN$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))))))
```

Finite Cocompleteness

SET.COL

Here we will present axioms that make the quasi-category of classes and functions finitely cocomplete. The finite colimits in the Classification (sub)Ontology use this.

The Namespace of Large Relations

This namespace will represent large binary relations and their morphisms. Some of the terms introduced in this namespace are listed in Table 1.

Table 1: Terms introduced into the large relation namespace

	Conglomerate	Function	Example, Relation
REL	'relation'	'class1', 'source', 'class2', 'target', 'extent' 'first', 'second' 'opposite', 'composition', 'identity' 'left-residuation', 'right-residuation' 'exponent', 'embed' 'fiber12', 'fiber21'	'subrelation'
REL .ENDO	'endorelation' 'reflexive' 'symmetric' 'antisymmetric' 'transitive' 'equivalence-relation'	'class', 'extent' 'opposite', 'composition', 'identity' 'binary-intersection' 'closure' 'equivalence-class', 'quotient', 'canon' 'equivalence-closure'	'subendorelation'

Table 2 lists the correspondence between standard mathematical notation and the ontological terminology in the namespace for binary relations.

Table 2: Correspondence between Mathematical Notation and Ontological Terminology

Mathematical Notation	Ontological Terminology	Natural Language Description
$\circ$	'REL$composition'	composition
$\backslash$	'REL$left-residuation'	left residuation
/	'REL$right-residuation'	right residuation
$(-)^{\infty}$ or $(-)^{\perp}$ or $(-)^{op}$	'REL$opposite'	involution – the opposite or dual relation

Relations

REL

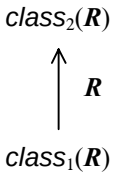


Figure 1: Large Binary Relation

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 $\boldsymbol{R} = \langle \mathit{class}_1(\boldsymbol{R}), \mathit{class}_2(\boldsymbol{R}), \mathit{extent}(\boldsymbol{R}) \rangle$ consists of a *first* class $\mathit{class}_1(\boldsymbol{R})$, a *second* class $\mathit{class}_2(\boldsymbol{R})$, and an *extent* class $\mathit{extent}(\boldsymbol{R}) \subseteq \mathit{class}_1(\boldsymbol{R}) \times \mathit{class}_2(\boldsymbol{R})$. We often use the following morphism notation for binary relations: $\boldsymbol{R} : \mathit{class}_1(\boldsymbol{R}) \to \mathit{class}_2(\boldsymbol{R})$.

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 first, second and extent classes.

```
(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)))
```

- Sometimes an alternate notation for the components is desired. This follows the morphism notation.

```
(5) (CNG$function source)
    (CNG$signature source relation SET$class)
    (= source class1)

(6) (CNG$function target)
    (CNG$signature target relation SET$class)
    (= target class2)
```

- Although not part of the basic definition of binary relations, there are two obvious projection functions from the extent to the component classes. These make relations into spans.

```
(7) (CNG$function first)
    (CNG$signature first relation SET.FTN$function)
    (forall (?r (relation ?r))
```

```
          (and (= (SET.FTN$source (first ?r)) (extent ?r))
               (= (SET.FTN$target (first ?r)) (class ?r))
               (forall (?x1 ?x2 ((extent ?r) [?x1 ?x2]))
                   (= ((first ?r) [?x1 ?x2]) ?x1))))

(8) (CNG$function second)
    (CNG$signature second relation SET.FTN$function)
    (forall (?r (relation ?r))
        (and (= (SET.FTN$source (second ?r)) (extent ?r))
             (= (SET.FTN$target (second ?r)) (class ?r))
             (forall (?x1 ?x2 ((extent ?r) [?x1 ?x2]))
                 (= ((second ?r) [?x1 ?x2]) ?x2))))
```

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

```
(9) (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* or *transpose 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.

```
(10) (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 target class of R is the same as the source class of S. There is a binary CNG function *composition* that takes two composable relations and returns their composition.

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

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

```
(12) (CNG$function identity)
     (CNG$signature identity class relation)
     (forall (?c (class ?c))
         (and (= (source (identity ?c)) ?c)
              (= (target (identity ?c)) ?c)
              (forall (?x1 (?c ?x1) ?x2 (?c ?x2))
                  (<=> ((extent (identity ?c)) [?x1 ?x2])
                       (= ?x1 ?x2))))))
```

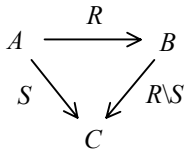


Figure 3: Left-Residuation

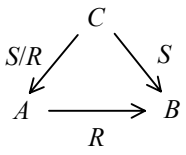


Figure 4: Right-Residuation

- Composition has an adjoint (generalized inverse) in two senses. Two relations *R* and *S* are *left residuable* when the first class of *R* is the same as the first class of *S*. There is a binary CNG function *left implication* that takes two left residuable relations and returns their left implication. Dually, two relations *R* and *S* are *right residuable* when the second class of *R* is the same as the second class of *S*. There is a binary CNG function *right implication* that takes two right residuable relations and returns their right implication.

```
(13) (CNG$function left-residuation)
     (CNG$signature left-residuation relation relation relation)
     (forall (?r (relation ?r) ?s (relation ?s))
         (<=> (exists (?t (relation ?t)) (= (left-residuation ?r ?s) ?t))
              (= (source ?r) (source ?s)))))
     (forall (?r (relation ?r) ?s (relation ?s))
         (=> (= (source ?r) (source ?s))
             (and (= (source (left-residuation ?r ?s)) (target ?r))
                  (= (target (left-residuation ?r ?s)) (target ?s)))))
     (forall (?r (relation ?r) ?s (relation ?s))
         (=> (= (source ?r) (source ?s))
             (forall (?y ((target ?r) ?y) ?z ((target ?s) ?z))
                 (<=> ((extent (left-residuation ?r ?s)) [?y ?z])
                      (forall (?x ((source ?r) ?x))
                          (=> ((extent ?r) [?x ?y]) ((extent ?s) [?x ?z])))))))

(14) (CNG$function right-residuation)
     (CNG$signature right-residuation relation relation relation)
     (forall (?r (relation ?r) ?s (relation ?s))
         (<=> (exists (?t (relation ?t)) (= (right-residuation ?r ?s) ?t))
              (= (target ?r) (target ?s)))))
     (forall (?r (relation ?r) ?s (relation ?s))
         (=> (= (target ?r) (target ?s))
             (and (= (source (right-residuation ?r ?s)) (source ?s))
                  (= (target (right-residuation ?r ?s)) (source ?r)))))
     (forall (?r (relation ?r) ?s (relation ?s))
         (=> (= (target ?r) (target ?s))
             (forall (?z ((source ?s) ?z) ?x ((source ?r) ?x))
                 (<=> ((extent (right-residuation ?r ?s)) [?z ?x])
                      (forall (?y ((target ?r) ?y))
                          (=> ((extent ?r) [?x ?y]) ((extent ?s) [?z ?y])))))))
```

- We can prove the theorem that left composition is (left) adjoint to left residuation:

  $R \circ T \subseteq S$ iff $T \subseteq R \backslash S$, for any compatible relations *R*, *S* and *T*.

  We can also prove the theorem that right composition is (left) adjoint to right residuation:

  $T \circ R \subseteq S$ iff $T \subseteq S/R$, for any compatible binary relations *R*, *S* and *T*.

```
(forall (?r (relation ?r) ?s (relation ?s) ?t (relation ?t))
    (=> (and (= (target ?r) (source ?t))
             (= (target ?s) (target ?t))
             (= (source ?r) (source ?s)))
        (<=> (subrelation (composition ?r ?t) ?s)
             (subrelation ?t (left-residuation ?r ?s)))))

(forall (?r (relation ?r) ?s (relation ?s) ?t (relation ?t))
    (=> (and (= (target ?t) (source ?r))
             (= (source ?t) (source ?s))
             (= (target ?r) (target ?s)))
        (<=> (subrelation (composition ?t ?r) ?s)
             (subrelation ?t (right-residuation ?r ?s)))))
```

- Residuation preserves composition

  $(R_1 \circ R_2)\backslash T = R_2\backslash(R_1\backslash T)$ and $T/(S_1 \circ S_2) = (T/S_2)/S_1$, for all compatible relations.

  Residuation preserves identity

  $Id_A\backslash T = T$ and $T/Id_B = T$, for all relations $T \subseteq A \times B$.

```
(forall (?r1 (relation ?r1) ?r2 (relation ?r2) ?t (relation ?t))
    (=> (and (= (target ?r1) (source ?r2))
             (= (source ?r1) (source ?t)))
        (= (left-residuation (composition ?r1 ?r2) ?t)
           (left-residuation ?r2 (left-residuation ?r1 ?t)))))

(forall (?s1 (relation ?s1) ?s2 (relation ?s2) ?t (relation ?t))
    (=> (and (= (target ?s1) (source ?s2))
             (= (target ?s2) (target ?t)))
        (= (right-residuation (composition ?s1 ?s2) ?t)
           (right-residuation ?s1 (right-residuation ?s2 ?t)))))

(forall (?t (relation ?t))
    (and (= (left-residuation (identity (source ?t)) ?t) ?t)
         (= (right-residuation (identity (target ?t)) ?t) ?t)))
```

- A theorem about transpose states that transpose dualizes residuation:

  $(R\backslash T)^{\infty} = T^{\infty}/R^{\infty}$ and $(T/S)^{\infty} = S^{\infty}\backslash T^{\infty}$.

```
(forall (?r (relation ?r) ?t (relation ?t))
    (=> (= (source ?r) (source ?t))
        (= (opposite (left-residuation ?r ?t))
           (right-residuation (opposite ?r) (opposite ?t)))))

(forall (?s (relation ?s) ?t (relation ?t))
    (=> (= (target ?s) (target ?t))
        (= (opposite (right-residuation ?s ?t))
           (left-residuation (opposite ?s) (opposite ?t)))))
```

- We can prove a general associative law:

  $(R\backslash T)/S = R\backslash(T/S)$, for all compatible relations $T \subseteq A \times B$, $R \subseteq A \times C$ and $S \subseteq D \times B$.

```
(forall (?r (relation ?r) ?s (relation ?s) ?t (relation ?t))
    (=> (and (= (source ?t) (source ?r))
             (= (target ?t) (target ?s)))
        (= (right-residuation ?s (left-residuation ?r ?t))
           (left-residuation ?r (right-residuation ?s ?t)))))
             (subrelation ?t (right-residuation ?r ?s)))))
```

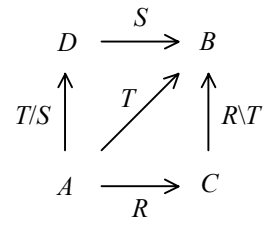


Figure 5: Associative law

- Functions have a special behavior with respect to derivation.
  If function f and relation R are composable, then

  $f \circ R = f^{\infty}\backslash R$.

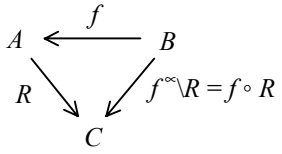


Figure 6: pre-composition

- If relation S and the opposite of function g are composable, then

  $S \circ g^{\infty} = S/g$.

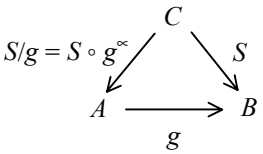


Figure 7: post-composition

```
(forall (?f (SET.FTN$function ?f) ?r (relation ?r))
    (=> (= (target ?f) (source ?r))
        (= (composition (SET.FTN$fn2rel ?f) ?r)
           (left-residuation (opposite (SET.FTN$fn2rel ?f)) ?r))))

(forall (?s (relation ?s) ?g (SET.FTN$function ?g))
    (=> (= (target ?s) (target ?g))
        (= (composition ?s (opposite (SET.FTN$fn2rel ?g)))
           (right-residuation (SET.FTN$fn2rel ?g) ?s))))
```

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

```
(15) (CNG$function exponent)
     (CNG$signature exponent SET$class SET$class SET$class)
     (forall (?c1 ?c2 (SET$class ?c1) (SET$class ?c2) ?r (REL$relation ?r))
        (<=> ((exponent ?c1 ?c2) ?r)
             (and (= (source ?r) ?c1)
                  (= (target ?r) ?c2))))
```

- For any class C there is a bijective function $embed_C : \wp C \to \mathsf{REL}[1, C]$.

```
(16) (CNG$function embed)
     (CNG$signature embed SET$class SET.FTN$function)
     (forall (?c (SET$class ?c))
         (and (= (SET.FTN$source (embed ?c)) (SET$power ?c))
              (= (SET.FTN$target (embed ?c)) (exponent SET.LIM$unit ?c))
              (forall (?b (SET$subclass ?b ?c) ?x (?c ?x))
                  (<=> ((extent ((embed ?c) ?b)) [0 ?x])
                       (?b ?x))))))
```

- For any binary relation $\boldsymbol{R} : X_1 \to X_2$ there are fiber functions $\phi^{\boldsymbol{R}}{}_{12} : X_1 \to \wp X_2$ and $\phi^{\boldsymbol{R}}{}_{21} : X_2 \to \wp X_1$ defined as follows.

$$\phi^{\boldsymbol{R}}{}_{12}(x_1) = \{x_2 \in X_2 \mid x_1 R x_2\}$$

$$\phi^{S}{}_{21}(x_2) = \{x_1 \in X_1 \mid x_1 R x_2\}$$

```
(17) (CNG$function fiber12)
     (CNG$signature fiber12 relation SET.FTN$function)
     (forall (?r) (relation ?r))
         (and (= (SET.FTN$source (fiber12 ?r)) (class1 ?r))
              (= (SET.FTN$target (fiber12 ?r)) (SET$power (class2 ?r)))
            (forall (?x1 ((class1 ?r) ?x1)
                     ?x2 ((class2 ?r) ?x1))
                (<=> (((fiber12 ?r) ?x1) ?x2)
                     ((extent ?r) [?x1 ?x2]))))))

(18) (CNG$function fiber21)
     (CNG$signature fiber21 relation SET.FTN$function)
     (forall (?r) (relation ?r))
         (and (= (SET.FTN$source (fiber21 ?r)) (class2 ?r))
              (= (SET.FTN$target (fiber21 ?r)) (SET$power (class1 ?r)))
            (forall (?x1 ((class1 ?r) ?x1)
                     ?x2 ((class2 ?r) ?x1))
                (<=> (((fiber21 ?r) ?x2) ?x1)
                     ((extent ?r) [?x1 ?x2]))))))
```

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)))
```

- Two endorelations R and S are *compatible* when the class of R is the same as the class of S. There is a binary CNG function *composition* that takes two compatible endorelations and returns their composition.

```
(5) (CNG$function composition)
    (CNG$signature composition endorelation endorelation endorelation)
    (forall (?r (endorelation ?r) ?s (endorelation ?s))
        (<=> (exists (?t (endorelation ?t)) (= (composition ?r ?s) ?t))
             (= (class ?r) (class ?s)))))
    (forall (?r (relation ?r) ?s (relation ?s))
        (=> (= (class ?r) (class ?s))
            (= (class (composition ?r ?s)) (class ?r))))
    (forall (?r (relation ?r) ?s (relation ?s))
        (=> (= (class ?r) (class ?s))
            (= (composition ?r ?s) (REL$composition ?r ?s))))
```

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

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

  - 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.

```
(7) (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))
```

- There is also a binary CNG function *binary-intersection* that takes two compatible endorelations and returns their intersection.

```
(8) (CNG$function binary-intersection)
    (CNG$signature binary-intersection endorelation endorelation endorelation)
    (forall (?r (endorelation ?r) ?s (endorelation ?s))
        (<=> (exists (?t (endorelation ?t)) (= (binary-intersection ?r ?s) ?t))
             (= (class ?r) (class ?s)))))
    (forall (?r (relation ?r) ?s (relation ?s))
        (=> (= (class ?r) (class ?s))
            (and (= (class (binary-intersection ?r ?s)) (class ?r))))
                 (= (extent (binary-intersection ?r ?s))
                    (SET$binary-intersection (extent ?r) (extent ?s))))))
```

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

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

- Or expressed more abstractly (without elements).

```
(10) (CNG$conglomerate reflexive)
     (CNG$subconglomerate reflexive endorelation)
     (forall (?r (endorelation ?r))
         (<=> (reflexive ?r)
              (subendorelation (identity (class ?r)) ?r)))
```

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

```
(11) (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]))))))
```

- Or expressed more abstractly (without elements).

```
(12) (CNG$conglomerate symmetric)
     (CNG$subconglomerate symmetric endorelation)
     (forall (?r (endorelation ?r))
         (<=> (symmetric ?r)
              (subendorelation (opposite ?r) ?r)))
```

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

```
(13) (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))))))
```

- Or expressed more abstractly (without elements).

```
(14) (CNG$conglomerate antisymmetric)
     (CNG$subconglomerate antisymmetric endorelation)
     (forall (?r (endorelation ?r))
         (<=> (antisymmetric ?r)
              (subendorelation
                  (binary-intersection (opposite ?r) ?r)
                  (identity (class ?r))))))
```

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

```
(15) (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])))))
```

- Or expressed more abstractly (without elements).

```
(16) (CNG$conglomerate transitive)
     (CNG$subconglomerate transitive endorelation)
     (forall (?r (endorelation ?r))
         (<=> (transitive ?r)
              (subendorelation (composition ?r ?r) ?r)))
```

- Any endorelation freely generates a preorder – the smallest preorder containing it called its reflexive-transitive *closure*.

```
(17) (CNG$function closure)
     (CNG$signature closure endorelation ORD$preorder)
     (forall (?r (endorelation ?r))
         (and (subendorelation ?r (closure ?r))
              (forall (?o (ORD$preorder ?o))
                  (=> (subendorelation ?r ?o)
                      (subendorelation (closure ?r) ?o)))))
```

- An *equivalence relation* ***E*** is a reflexive, symmetric and transitive endorelation. An equivalence relation determines a *quotient* class and a *canon*(ical) surjection. The canon is the factorization of the equivalence-class function through the quotient class. Every endorelation ***R*** generates an equivalence relation, the smallest equivalence relation containing it. This is the reflexive, symmetric, transitive closure of ***R***.

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

(19) (CNG$function equivalence-class)
     (CNG$signature equivalence-class equivalence-relation SET.FTN$function)
     (forall (?e (equivalence-relation ?e))
         (and (SET.FTN$source (equivalence-class ?e) (class ?e))
              (SET.FTN$target (equivalence-class ?e) (SET$power (class ?e)))
              (forall (?x1 ((class ?e) ?x)) ?x2 ((class ?e) ?x2))
                  (<=> (((equivalence-class ?e) ?x1) ?x2)
                       ((extent ?e) [?x1 ?x2])))))

(20) (CNG$function quotient)
     (CNG$signature quotient equivalence-relation class)
     (forall (?e (equivalence-relation ?e))
         (= (quotient ?e)
            (SET.FTN$image (equivalence-class ?e))))

(21) (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))))

(22) (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)))))))
```

The Classification Ontology

The Namespace of Large Orders

This namespace will represent large orders and their morphisms: monotonic functions, adjoint pairs and Galois connections. Some of the terms introduced in this namespace are listed in Table 1. The *italicized terms* below remain to be defined.

Table 1: Terms introduced into the large order namespace

	Conglomerate	Function	Example
ORD	'preorder' 'partial-order' 'total-order'	'class', 'underlying', 'extent' '**classification**' 'identity', 'power' 'up-class', 'down-class' 'up', 'down' (uses ∃ quantifier) 'up-embedding', 'down-embedding' 'greatest', 'least'	'successor' 'natural-numbers-order'
ORD .DM		'upper-bound', 'lower-bound' (uses ∀ quantifier) '*upper-lower-closure*', '*lower-upper-closure*' '*cut*', '*cut-down*', '*cut-up*' '*cut-order*', '*meet*', '*join*' 'join-dense', 'meet-dense' '***complete-lattice***' '*element-embedding*', '*element-concept*' '***dedekind-macneille***'	
ORD .FTN	'monotonic-function'	'source', 'target', 'function' 'opposite', 'composition', 'identity' 'left', 'right'	
ORD .BIMOD	'bimodule'	'source', 'target', 'relation' 'opposite', 'composition', 'identity'	
ORD .ADJ	'adjoint-pair'	'source', 'target', 'left', 'right' '**infomorphism**', '**bond**' 'opposite', 'composition', 'identity'	
ORD .GC	'galois-connection'	'source', 'target', 'left', 'right' 'source-closure', 'target-closure'	

Table 2 lists (needs to be completed) the correspondence between standard mathematical notation and the ontological terminology in the order namespace.

Table 2: Correspondence between Mathematical Notation and Ontological Terminology

Math	Ontological Terminology	Natural Language Description
≤	'ORD$extent'	the order relation class of a preorder

Orders

ORD

- A *preorder* $\boldsymbol{A} = \langle A, \leq_A \rangle = \langle \mathit{class}(\boldsymbol{A}), \mathit{ext}(\boldsymbol{A}) \rangle$ is a reflexive and transitive endorelation. For convenience of reference the class terms $A = \mathit{class}(\boldsymbol{A})$ and $\leq_A = \mathit{ext}(\boldsymbol{A})$ are reissued here.

```
(1) (CNG$conglomerate preorder)
    (CNG$subconglomerate preorder REL.ENDO$endorelation)

(2) (CNG$function class)
    (CNG$function underlying)
    (= underlying class)
    (CNG$signature class preorder SET$class)
    (forall (?o (preorder ?o))
        (= (class ?o) (REL.ENDO$class ?o)))

(3) (CNG$function extent)
    (CNG$signature extent preorder SET$class)
    (forall (?o (preorder ?o))
        (= (extent ?o) (REL.ENDO$extent ?o)))

    (forall (?o (preorder ?o))
        (and (reflexive ?o) (transitive ?o)))
```

- Any preorder $\boldsymbol{A} = \langle A, \leq_A \rangle = \langle \mathit{class}(\boldsymbol{A}), \mathit{ext}(\boldsymbol{A}) \rangle$ has an associated classification $\mathit{cls}(\boldsymbol{A}) = \langle \mathit{class}(\boldsymbol{A}), \mathit{class}(\boldsymbol{A}), \mathit{ext}(\boldsymbol{A}) \rangle$. In axiom (4) the CNG function '`classification`' represents this.

```
(4) (CNG$function classification)
    (CNG$signature classification preorder CLS$classification)
    (forall (?o (preorder ?o))
        (and (= (instance (classification ?o)) (class ?o))
             (= (type (classification ?o)) (class ?o))
             (= (incidence (classification ?o)) (extent ?o))))
```

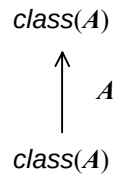


Figure 1: Preorder as Classification

- A *partial order E* is a preorder that is antisymmetric.

```
(5) (CNG$conglomerate partial-order)
    (CNG$subconglomerate partial-order preorder)
    (forall (?o (REL.ENDO$endorelation ?o))
        (<=> (partial-order ?o)
             (and (preorder ?o) (antisymmetric ?o))))
```

- The identity endorelation on any class A is a partial order. The CNG function in axiom (6) realizes this.

```
(6) (CNG$function identity)
    (CNG$signature identity class partial-order)
    (forall (?c (class ?c))
        (= (identity ?c) (REL.ENDO$identity ?c)))
```

  - For any class C the power class $\wp(C)$ using the subclass ordering is a partial order. There is a CNG function power : class $\rightarrow$ order that maps a class to its power order.

```
(7) (CNG$function power)
    (CNG$signature power class partial-order)
    (forall (?c (class ?c))
        (and (= (class (power ?c)) (SET$power ?c))
             (forall (?c1 (SET$subclass ?c1 ?c)
                      ?c2 (SET$subclass ?c2 ?c))
                 (<=> ((extent (power ?c)) [?c1 ?c2])
                      (SET$subclass ?c1 ?c2))))))
```

- A *total order E* is a partial order where all pairs are comparable.

```
(8) (CNG$conglomerate total-order)
    (CNG$subconglomerate total-order partial-order)
    (forall (?o (REL.ENDO$endorelation ?o))
        (<=> (total-order ?o)
```

```
            (and (partial-order ?o)
                 (forall (?x1 ((class ?o) ?x1)
                          ?x2 ((class ?o) ?x2))
                     (or ((extent ?o) [?x1 ?x2])
                         ((extent ?o) [?x2 ?x1])))))
```

- The natural numbers object in the core namespace has a "naturally defined" preorder. We can prove that this is a total order.

```
(9) (REL.ENDO$endorelation successor)
    (= (REL.ENDO$class successor) SET.TOP$natural-numbers)
    (<=> (REL.ENDO$extent successor) [?m ?n])
         (= (SET.TOP$successor ?m) ?n))

(10) (partial-order natural-numbers-order)
     (= natural-numbers-order (closure successor))

     (total-order natural-numbers-order)
```

- Let $\boldsymbol{P} = \langle P, \leq \rangle$ be a partial order and let $Q \subseteq P$. Q is a *down-class* (*deceasing class* or *order ideal*) if, whenever $x \in Q$, $y \in P$ and $y \leq x$, we have $y \in Q$. An *up-class* (*increasing class* or *order filter*) has a dual definition.

```
(11) (KIF$function down-class)
     (KIF$signature down-class partial-order CNG$conglomerate)
     (forall (?o (partial-order ?o)
              ?c (SET$subclass ?c (class ?o)))
         (<=> ((down-class ?o) ?c)
              (forall (?x (?c ?x) ?y (?o ?y))
                  (=> ((extent ?o) [?y ?x]) (?c ?y)))))

(12) (KIF$function up-class)
     (KIF$signature up-class partial-order CNG$conglomerate)
     (forall (?o (partial-order ?o)
              ?c (SET$subclass ?c (class ?o)))
         (<=> ((up-class ?o) ?c)
              (forall (?x (?c ?x) ?y (?o ?y))
                  (=> ((extent ?o) [?x ?y]) (?c ?y)))))
```

- Let $\boldsymbol{P} = \langle P, \leq \rangle$ be a partial order and let $Q \subseteq P$. The class $\uparrow_{\boldsymbol{P}}(Q) = \mathit{up}_{\boldsymbol{P}}(Q)$ is the class of all $\boldsymbol{P}$-elements that are above <u>some</u> (existential quantifier) $\boldsymbol{Q}$-element. Dually, the class $\downarrow_{\boldsymbol{P}}(Q) = \mathit{down}_{\boldsymbol{P}}(Q)$ is the class of all $\boldsymbol{P}$-elements that are below <u>some</u> (existential quantifier) $\boldsymbol{Q}$-element.

```
(13) (CNG$function up)
     (CNG$signature up partial-order SET.FTN$function)
     (forall (?o (partial-order ?o))
         (and (SET.FTN$source (up ?o)) (SET$power (class ?o)))
              (SET.FTN$target (up ?o)) (SET$power (class ?o)))
              (forall (?q (SET$subclass ?q (class ?o)) ?x ((class ?o) ?x))
                  (<=> (((up ?o) ?q) ?x)
                       (exists (?y (?q ?y))
                           ((extent ?o) [?y ?x])))))

(14) (CNG$function down)
     (CNG$signature down partial-order SET.FTN$function)
     (forall (?o (partial-order ?o))
         (and (SET.FTN$source (down ?o)) (SET$power (class ?o)))
              (SET.FTN$target (down ?o)) (SET$power (class ?o)))
              (forall (?q (SET$subclass ?q (class ?o)) ?x ((class ?o) ?x))
                  (<=> (((down ?o) ?q) ?x)
                       (exists (?y (?q ?y))
                           ((extent ?o) [?x ?y])))))
```

- It is easy to check that $\downarrow_{\boldsymbol{P}}(Q)$ is the smallest down-class containing Q and that Q is a down-class iff $Q = \downarrow_{\boldsymbol{P}}(Q)$. Dually for $\uparrow_{\boldsymbol{P}}(Q)$.

```
     (forall (?o (partial-order ?o)
              ?q (SET$subclass ?q (class ?o)))
         (and ((up-class ?o) ((up ?o) ?q))
              (SET$subclass ?q ((up ?o) ?q))
```

```
            (forall (?c (SET$subclass ?c (class ?o)))
                (=> (and ((up-class ?o) ?c) (SET$subclass ?q ?c))
                    (SET$subclass ((up ?o) ?q) ?c)))))

    (forall (?o (partial-order ?o)
            ?q (SET$subclass ?q (class ?o)))
        (<=> ((up-class ?o) ?q)
            (= ?q ((up ?o) ?q))))

    (forall (?o (partial-order ?o)
            ?q (SET$subclass ?q (class ?o)))
        (and ((down-class ?o) ((down ?o) ?q))
            (SET$subclass ?q ((down ?o) ?q))
            (forall (?c (SET$subclass ?c (class ?o)))
                (=> (and ((down-class ?o) ?c) (SET$subclass ?q ?c))
                    (SET$subclass ((down ?o) ?q) ?c)))))

    (forall (?o (partial-order ?o)
            ?q (SET$subclass ?q (class ?o)))
        (<=> ((down-class ?o) ?q)
            (= ?q ((down ?o) ?q))))
```

- Let $\boldsymbol{P} = \langle P, \leq \rangle$ be a partial order and let $q \in P$. The class $\uparrow_{\boldsymbol{P}} q = \mathit{up\text{-}embed}_{\boldsymbol{P}}(q)$ is the class of all $\boldsymbol{P}$-elements that are above q. Dually, the class $\downarrow_{\boldsymbol{P}} q = \mathit{down\text{-}embed}_{\boldsymbol{P}}(q)$ is the class of all $\boldsymbol{P}$-elements that are below q.

```
(15) (CNG$function up-embedding)
     (CNG$signature up-embedding partial-order SET.FTN$function)
     (forall (?o (partial-order ?o))
         (and (SET.FTN$source (up-embedding ?o)) (class ?o))
              (SET.FTN$target (up-embedding ?o)) (SET$power (class ?o)))
              (forall (?q ((class ?o) ?q) ?y ((class ?o) ?y))
                  (<=> (((up-embedding ?o) ?q) ?y)
                           ((extent ?o) [?q ?y])))))

(16) (CNG$function down-embedding)
     (CNG$signature down-embedding partial-order SET.FTN$function)
     (forall (?o (partial-order ?o))
         (and (SET.FTN$source (down-embedding ?o)) (class ?o))
              (SET.FTN$target (down-embedding ?o)) (SET$power (class ?o)))
              (forall (?q ((class ?o) ?q) ?x ((class ?o) ?x))
                  (<=> (((down-embedding ?o) ?q) ?x)
                           ((extent ?o) [?x ?q])))))
```

- It is easy to check that $\downarrow_{\boldsymbol{P}}\{p\} = \downarrow_{\boldsymbol{P}} p$. Dually for $\uparrow_{\boldsymbol{P}} q$.

```
    (forall (?o (partial-order ?o)
        (and (= (SET.FTN$composition (SET.FTN$singleton (class ?o)) (down ?o))
                (down-embedding ?o))
             (= (SET.FTN$composition (SET.FTN$singleton (class ?o)) (up ?o))
                (up-embedding ?o))))
```

- Let $\boldsymbol{P} = \langle P, \leq \rangle$ be a partial order and let $Q \subseteq P$. An element $a \in Q$ is a *greatest* (or *maximum*) element of Q if $x \leq a$ for any element $x \in Q$. Dually, an element $a \in Q$ is a *least* (or *minimum*) element of Q if $a \leq x$ for any element $x \in Q$.

```
(17) (KIF$function greatest)
     (KIF$signature greatest partial-order CNG$function)
     (forall (?o (partial-order ?o))
         (and (CNG$signature (greatest ?o) SET$class SET$class)
              (forall (?c (SET$subclass ?c (class ?o))
                       ?a (?c ?a))
                  (<=> (((greatest ?o) ?c) ?a)
                       (forall (?x (?c ?x))
                           ((extent ?o) [?x ?a])))))))

(18) (KIF$function least)
     (KIF$signature least partial-order CNG$function)
     (forall (?o (partial-order ?o))
         (and (CNG$signature (least ?o) SET$class SET$class)
```

```
            (forall (?c (SET$subclass ?c (class ?o))
                     ?a (?c ?a))
                (<=> (((least ?o) ?c) ?a)
                     (forall (?x (?c ?x))
                         ((extent ?o) [?a ?x])))))
```

- By antisymmetry, there is at most one greatest or least element. Therefore, we can prove the following theorems.

```
(forall (?o (partial-order ?o)
         ?c (SET$subclass ?c (class ?o))
         ?x (((greatest ?o) ?c) ?x)
         ?y (((greatest ?o) ?c) ?y))
    (= ?x ?y))

(forall (?o (partial-order ?o)
         ?c (SET$subclass ?c (class ?o))
         ?x (((least ?o) ?c) ?x)
         ?y (((least ?o) ?c) ?y))
    (= ?x ?y))
```

Dedekind-MacNeille Completion

ORD.DM

- Let $\boldsymbol{P} = \langle P, \leq \rangle$ be a partial order and let $Q \subseteq P$. An element $x \in P$ is an *upper bound* of Q when $q \leq x$ for all (universal quantifier) $q \in Q$. A *lower bound* is defined dually. The set of all upper bounds of Q is denoted Q^{u}. Dually, the set of all lower bounds of Q is denoted Q^{l}. These classes may be empty.

```
(1) (CNG$function upper-bound)
    (CNG$signature upper-bound ORD$partial-order SET.FTN$function)
    (forall (?o (ORD$partial-order ?o))
        (and (SET.FTN$source (upper-bound ?o)) (SET$power (class ?o)))
             (SET.FTN$target (upper-bound ?o)) (SET$power (class ?o)))
             (forall (?q (SET$subclass ?q (ORD$class ?o)) ?y ((ORD$class ?o) ?y))
                 (<=> (((upper-bound ?o) ?q) ?y)
                      (forall (?x (?q ?x))
                          ((ORD$extent ?o) [?x ?y])))))))

(2) (CNG$function lower-bound)
    (CNG$signature lower-bound ORD$partial-order SET.FTN$function)
    (forall (?o (ORD$partial-order ?o))
        (and (SET.FTN$source (lower-bound ?o)) (SET$power (class ?o)))
             (SET.FTN$target (lower-bound ?o)) (SET$power (class ?o)))
             (forall (?q (SET$subclass ?q (ORD$class ?o)) ?x ((ORD$class ?o) ?x))
                 (<=> (((lower-bound ?o) ?q) ?x)
                      (forall (?y (?q ?y))
                          ((ORD$extent ?o) [?x ?y])))))))
```

- Since the order is transitive, the upper bound of Q is always an up-class and the lower bound of Q is always a down-class.

```
    (forall (?o (ORD$partial-order ?o)
             ?q (SET$subclass ?q (ORD$class ?o)))
        (and ((ORD$up-class ?o) ((upper-bound ?o) ?q))
             ((ORD$down-class ?o) ((lower-bound ?o) ?q))))
```

- It is easy to check that $\{p\}^{\mathsf{l}} = \downarrow_{\boldsymbol{P}} p$. Dually, $\{p\}^{\mathsf{u}} = \uparrow_{\boldsymbol{P}} p$.

```
    (forall (?o (partial-order ?o))
        (and (= (SET.FTN$composition (SET.FTN$singleton (class ?o)) (upper-bound ?o))
                (up-embedding ?o))
             (= (SET.FTN$composition (SET.FTN$singleton (class ?o)) (lower-bound ?o))
                (down-embedding ?o))))
```

- The composition of bounds gives two senses of closure operator.

```
(3) (CNG$function upper-lower-closure)
    (CNG$signature upper-lower-closure ORD$partial-order SET.FTN$function)
    (forall (?o (ORD$partial-order ?o))
        (and (= (SET.FTN$source (upper-lower-closure ?o))
```

```
            (SET$power (ORD$class ?o)))
         (= (SET.FTN$target (upper-lower-closure ?o))
            (SET$power (ORD$class ?o)))
         (= (upper-lower-closure ?o)
            (SET.FTN$composition (upper-bound ?o) (lower-bound ?o)))))

(4) (CNG$function lower-upper-closure)
    (CNG$signature lower-upper-closure ORD$partial-order SET.FTN$function)
    (forall (?o (ORD$partial-order ?o))
        (and (= (SET.FTN$source (lower-upper-closure ?o))
                (SET$power (ORD$class ?o)))
             (= (SET.FTN$target (lower-upper-closure ?o))
                (SET$power (ORD$class ?o)))
             (= (lower-upper-closure ?o)
                (SET.FTN$composition (lower-bound ?o) (upper-bound ?o)))))
```

- The following (easily proven) results confirm that there is a Galois connection between bound operators and that the composites are closure operators.

  $X \subseteq X^{ul}$. and $X^{u} = X^{ulu}$ for all $X \subseteq \mathit{inst}(\boldsymbol{A})$.

  If $X \subseteq Y$ then $X^{u} \supseteq Y^{u}$ and $X^{l} \supseteq Y^{l}$.

  $Y \subseteq Y^{lu}$ and $Y^{l} = Y^{lul}$ for all $Y \subseteq \mathit{typ}(\boldsymbol{A})$.

```
(forall (?o (ORD$partial-order ?o))
    (and (= (upper-bound ?o)
            (SET.FTN$composition (upper-lower-closure ?o) (upper-bound ?o)))
         (forall (?x (SET$subclass ?x (ORD$class ?o)))
             (SET$subclass ?x ((upper-lower-closure ?o) ?x)))))

(forall (?o (ORD$partial-order ?o))
         ?x (SET$subclass ?x (ORD$class ?o))
         ?y (SET$subclass ?y (ORD$class ?o)))
    (=> (SET$subclass ?x ?y)
        (and (SET$subclass ((upper-bound ?o) ?y) ((upper-bound ?o) ?x))
             (SET$subclass ((lower-bound ?o) ?y) ((lower-bound ?o) ?x)))))

(forall (?o (ORD$partial-order ?o))
    (and (= (lower-bound ?o)
            (SET.FTN$composition (lower-upper-closure ?o) (lower-bound ?o)))
         (forall (?y (SET$subclass ?y (ORD$class ?o)))
             (SET$subclass ?y ((lower-upper-closure ?o) ?y)))))
```

- Further properties of Galois connection relate to continuity – the closure of a union of a family of classes is the intersection of the closures.

  $(\cup_{j \in J} X_j)^{u} = \cap_{j \in J} X_j^{u}$ for any family of subsets $X_j \subseteq \mathit{inst}(\boldsymbol{A})$ for $j \in J$.

  $(\cup_{k \in K} Y_k)^{l} = \cap_{k \in K} Y_k^{l}$ for any family of subsets $Y_k \subseteq \mathit{typ}(\boldsymbol{A})$ for $k \in K$.

- It is important to note that the notions of upper bound and lower bound for orders are related to the notions of intent and extent for classifications.
  - The up operator is the left-derivation of the classification of a preorder, and the down operator is the left-derivation of the classification of a preorder.
  - The cut class is the concept class of the classification of a preorder.
  - The Dedekind-MacNeille completion of a partial order is the concept lattice of the classification of a preorder, but with the instance/type classes identified with the preorder class, and the instance/type embeddings identified with the preorder embedding.

```
(forall (?o (partial-order ?o)
         ?q (SET$subclass ?q (class ?o)))
    (and ((up-class ?o) ((upper-bound ?o) ?q))
         ((down-class ?o) ((lower-bound ?o) ?q))))
```

- Let $\boldsymbol{P} = \langle P, \leq \rangle$ be a partial order and let $Q \subseteq P$. Then Q is *join-dense* in P when for every element $a \in P$ there is a subset $X \subseteq Q$ such that $a = \sqcup_P(X)$. The notion of *meet-dense* is dual.

```
(21) (KIF$function join-dense)
     (KIF$signature join-dense partial-order SET$class)
     (forall (?o (partial-order ?o))
         (and (SET$subclass (join-dense ?o) (SET$power (class ?o)))
              (forall (?q (SET$subclass ?q (class ?o)))
                  (<=> ((join-dense ?o) ?q)
                       (forall (?a ((class ?o) ?a))
                           (exists (?x (SET$subclass ?x ?q))
                                (((least ?o) ((upper-bound ?o) ?x)) ?a)))))))

(22) (KIF$function meet-dense)
     (KIF$signature meet-dense partial-order SET$class)
     (forall (?o (partial-order ?o))
         (and (SET$subclass (meet-dense ?o) (SET$power (class ?o)))
              (forall (?q (SET$subclass ?q (class ?o)))
                  (<=> ((meet-dense ?o) ?q)
                       (forall (?a ((class ?o) ?a))
                           (exists (?x (SET$subclass ?x ?q))
                                (((greatest ?o) ((lower-bound ?o) ?x)) ?a)))))))
```

- Let $\boldsymbol{L} = \langle L, \leq_L \rangle$ be a partial order. For any class $S \subseteq L$, if it exists, the *meet* (*greatest lower bound* or *infimum*) $\sqcap_L(S)$ is the greatest element in the lower bound of S. For any class $S \subseteq P$, if it exists, the *join* (*least upper bound* or *infimum*) $\sqcup_L(S)$ is the least element in the upper bound of S.

Monotonic Functions

ORD.FTN

Preorders and partial orders are related through monotonic functions.

- A monotonic function $f : \boldsymbol{P} \to \boldsymbol{Q}$, from preorder $\boldsymbol{P} = \langle P, \leq_{\boldsymbol{P}} \rangle$ to preorder $\boldsymbol{Q} = \langle Q, \leq_{\boldsymbol{Q}} \rangle$, is a function $f : P \to Q$ between the underlying classes that preserves order:

  if $p_1 \leq_{\boldsymbol{P}} p_2$ then $f(p_1) \leq_{\boldsymbol{Q}} f(p_2)$ for all $p_1, p_2 \in P$.

```
(1) (CNG$conglomerate monotonic-function)

(2) (CNG$function source)
    (CNG$signature source monotonic-function ORD$preorder)

(3) (CNG$function target)
    (CNG$signature target monotonic-function ORD$preorder)

(4) (CNG$function function)
    (CNG$signature function monotonic-function SET.FTN$function)
    (forall (?f (monotonic-function ?f))
        (and (= (SET.FTN$source (function ?f)) (ORD$class (source ?f)))
             (= (SET.FTN$target (function ?f)) (ORD$class (target ?f)))))

(5) (forall (?f (monotonic-function ?f)
             ?p1 ((ORD$class (source ?f)) ?p1)
             ?p2((ORD$class (source ?f)) ?p2)
        (=> ((ORD$extent (source ?f)) [?p1 ?p2])
            ((ORD$extent (target ?f)) [(?f ?p1) (?f ?p2)]))))
```

- Two monotonic functions are composable when the target preorder of the first is the source preorder of the second. The *composition* of two composable monotonic functions $f : \boldsymbol{P} \to \boldsymbol{Q}$ and $g : \boldsymbol{Q} \to \boldsymbol{N}$ is defined via the composition of the underlying functions.

```
(6) (CNG$function composition)
    (CNG$signature composition monotonic-function monotonic-function monotonic-function)
    (forall (?f (monotonic-function ?f) ?g (monotonic-function ?g))
        (<=> (exists (?h (monotonic-function ?h)) (= (composition ?f ?g) ?h))
             (= (target ?f) (source ?g))))
    (forall (?f (monotonic-function ?f) ?g (monotonic-function ?g))
        (=> (= (target ?f) (source ?g))
            (and (= (source (composition ?f ?g)) (source ?f))
                 (= (target (composition ?f ?g)) (target ?g))
                 (= (function (composition ?f ?g))
                    (SET.FTN$composition (function ?f) (function ?g))))))
```

- The identity monotonic function at a preorder $\boldsymbol{P}$ is the identity function of the underlying class.

```
(7) (CNG$function identity)
    (CNG$signature identity ORD$preorder monotonic-function)
    (forall (?p (ORD$preorder ?p))
        (and (= (source (identity ?p)) ?p)
             (= (target (identity ?p)) ?p)
             (= (function (identity ?p)) (SET.FTN$identity (ORD$underlying ?p)))))
```

- Duality can be extended from orders to monotonic functions. For any monotonic function $f : \boldsymbol{P} \to \boldsymbol{Q}$, the *opposite* or *dual* of f is the monotonic function $f^{\perp} : \boldsymbol{P}^{\perp} \to \boldsymbol{Q}^{\perp}$, whose source preorder is the opposite of the source of f, and whose target preorder is the opposite of the target of f. Note that the source/target polarity has not changed.

  Axiom (8) specifies the opposite operator on monotonic functions.

```
(8) (CNG$function opposite)
    (CNG$signature opposite monotonic-function monotonic-function)
    (forall (?f (monotonic-function ?f))
        (and (= (source (opposite ?f)) (ORD$opposite (source ?f)))
             (= (target (opposite ?f)) (ORD$opposite (target ?f)))
             (= (function (opposite ?f)) (function ?f))))
```

- In the presence of a preorder $\boldsymbol{A} = \langle A, \leq_A \rangle$, there are two ways that monotonic functions are transformed into order bimodules – both by composition. For any monotonic function $f : \boldsymbol{B} \to \boldsymbol{A}$, the *left* bimodule $f_@ : A \to B$ is defined as

  $$f_@(a, b) \text{ iff } a \leq_A f(b),$$

  and the *right* bimodule $f^@ : B \to A$ as follows

  $$f^@(b, a) \text{ iff } f(b) \leq_A a.$$

  The left and right operators for monotonic functions are defined in terms of the left and right operators for ordinary functions.

```
(9) (KIF$function left)
    (KIF$signature left preorder CNG$function)
    (forall (?o (ORD$preorder ?o))
        (CNG$signature (left ?o) monotonic-function BIMOD$bimodule))
    (forall (?o (ORD$preorder ?o)
             ?f (monotonic-function ?f))
        (<=> (exists (?r (BIMOD$bimodule ?r)) (= ((left ?o) ?f) ?r))
             (= (target ?f) ?o)))
    (forall (?o (ORD$preorder ?o) ?f (monotonic-function ?f))
        (=> (= (target ?f) ?o)
            (and (= (BIMOD$source ((left ?o) ?f)) ?o)
                 (= (BIMOD$target ((left ?o) ?f)) (source ?f))
                 (= (BIMOD$relation ((left ?o) ?f))
                    ((SET.FTN$left ?o) (function ?f))))))

(10) (KIF$function right)
     (KIF$signature right ORD$preorder CNG$function)
     (forall (?o (ORD$preorder ?o))
         (CNG$signature (right ?o) monotonic-function BIMOD$bimodule))
     (forall (?o (ORD$preorder ?o)
              ?f (monotonic-function ?f))
         (<=> (exists (?r (BIMOD$bimodule ?r)) (= ((right ?o) ?f) ?r))
              (= (target ?f) ?o)))
     (forall (?o (ORD$preorder ?o) ?f (monotonic-function ?f))
         (=> (= (target ?f) ?o)
             (and (= (BIMOD$source ((right ?o) ?f)) (source ?f))
                  (= (BIMOD$target ((right ?o) ?f)) ?o)
                  (= (BIMOD$relation ((right ?o) ?f))
                     ((SET.FTN$right ?o) (function ?f))))))
```

Order Bimodules

ORD.BIMOD

Preorders and partial orders are related through order bimodules. Order bimodules extend binary relations to the order realm. An order bimodule is a binary relation, whose source and target are preorders, and which is order-closed on the left (at the source) and on the right (at the target).

- A(n *order*) *bimodule* $\boldsymbol{R} : \boldsymbol{P} \to \boldsymbol{Q}$, from preorder $\boldsymbol{P} = \langle P, \leq_{\boldsymbol{P}} \rangle$ to preorder $\boldsymbol{Q} = \langle Q, \leq_{\boldsymbol{Q}} \rangle$, is a binary relation function $\boldsymbol{R} : P \to Q$ between the underlying classes that is order-closed on left and right:

  if $p_2 \leq_{\boldsymbol{P}} p_1$ and then $p_1 R q$ for all $p_2 R q$, and

  if $p R q_1$ and $q_1 \leq_{\boldsymbol{Q}} q_2$ then for all $p R q_2$.

```
(1) (CNG$conglomerate bimodule)

(2) (CNG$function source)
    (CNG$signature source bimodule ORD$preorder)

(3) (CNG$function target)
    (CNG$signature target bimodule ORD$preorder)

(4) (CNG$function relation)
    (CNG$signature relation bimodule REL$relation)
    (forall (?r (bimodule ?r))
        (and (= (REL$source (relation ?r)) (ORD$class (source ?r)))
             (= (REL$target (relation ?r)) (ORD$class (target ?r)))))

(5) (forall (?r (bimodule ?r)
             ?p2 ((ORD$class (source ?r)) ?p2)
             ?p1 ((ORD$class (source ?r)) ?p1)
             ?q ((ORD$class (target ?r)) ?q))
        (=> (and ((ORD$extent (source ?r)) [?p2 ?p1])
                 ((ORD$extent (relation ?r)) [?p1 ?q]))
            ((ORD$extent (relation ?r)) [?p2 ?q])))

    (forall (?r (bimodule ?r)
             ?p ((ORD$class (source ?r)) ?p)
             ?q1 ((ORD$class (target ?r)) ?q1)
             ?q2 ((ORD$class (target ?r)) ?q2))
        (=> (and ((ORD$extent (relation ?r)) [?p ?q1]))
                 ((ORD$extent (target ?r)) [?q1 ?q2])
            ((ORD$extent (relation ?r)) [?p ?q2])))
```

- Duality can be extended from monotonic functions to order bimodules. For any order bimodule $\boldsymbol{R} : \langle P, \leq_{\boldsymbol{P}} \rangle \to \langle Q, \leq_{\boldsymbol{Q}} \rangle$, from preorder $\boldsymbol{P} = \langle P, \leq_{\boldsymbol{P}} \rangle$ to preorder $\boldsymbol{Q} = \langle Q, \leq_{\boldsymbol{Q}} \rangle$, the *opposite* (*dual* or *transpose*) of $\boldsymbol{R}$ is the order bimodule $\boldsymbol{R}^{\perp} : \langle Q, \geq_{\boldsymbol{Q}} \rangle \to \langle P, \geq_{\boldsymbol{P}} \rangle$, from preorder $\boldsymbol{Q}^{\perp} = \langle Q, \geq_{\boldsymbol{Q}} \rangle$ to preorder $\boldsymbol{P}^{\perp} = \langle P, \geq_{\boldsymbol{P}} \rangle$, whose source preorder is the opposite of the target of $\boldsymbol{R}$, whose target preorder is the opposite of the source of $\boldsymbol{R}$, and whose underlying relation is the transpose (opposite) of the relation of $\boldsymbol{R}$. Note that the source/target polarity has changed – source to target and target to source.

  Axiom (6) specifies the opposite operator on order bimodules.

```
(6) (CNG$function opposite)
    (CNG$signature opposite bimodule bimodule)
    (forall (?r (bimodule ?r))
        (and (= (source (opposite ?r)) (ORD$opposite (target ?r)))
             (= (target (opposite ?r)) (ORD$opposite (source ?r)))
             (= (relation (opposite ?r)) (ORD$opposite (relation ?r)))))
```

- Two order bimodules $\boldsymbol{R} : \boldsymbol{O} \to \boldsymbol{P}$ and $\boldsymbol{S} : \boldsymbol{P} \to \boldsymbol{Q}$ are *composable* when the target order of $\boldsymbol{R}$ is the same as the source order of $\boldsymbol{S}$. There is a binary CNG function *composition*, which takes two composable order bimodules and returns their composition.

```
(7) (CNG$function composition)
```

```
(CNG$signature composition bimodule bimodule bimodule)
 (forall (?r (bimodule ?r) ?s (bimodule ?s))
     (<=> (exists (?t (bimodule ?t)) (= (composition ?r ?s) ?t))
          (= (target ?r) (source ?s)))))
 (forall (?r (bimodule ?r) ?s (bimodule ?s))
     (=> (= (target ?r) (source ?s))
         (and (= (source (composition ?r ?s)) (source ?r))
              (= (target (composition ?r ?s)) (target ?s))
              (= (relation (composition ?r ?s))
                 (REL$composition (relation ?r) (relation ?s))))))
```

- For any preorder ***P*** there is an identity order bimodule $identity_{P}$.

```
(8) (CNG$function identity)
    (CNG$signature identity ORD$preorder bimodule)
    (forall (?o (ORD$preorder ?o))
        (and (= (source (identity ?o)) ?o)
             (= (target (identity ?o)) ?o)
             (= (relation (identity ?o))
                (REL$identity (ORD$class ?o)))))
```

Adjoint Pairs

ORD.ADJ

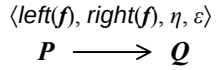


Figure 1: Abstract Notation

- An *adjoint pair* $\boldsymbol{f} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ from preorder $\boldsymbol{P}$ to preorder $\boldsymbol{Q}$ is a pair $\boldsymbol{f} = \langle \mathit{right}(\boldsymbol{f}), \mathit{left}(\boldsymbol{f}) \rangle$ of oppositely directed monotonic functions, $\mathit{right}(\boldsymbol{f}) : \boldsymbol{Q} \to \boldsymbol{P}$ and $\mathit{left}(\boldsymbol{f}) : \boldsymbol{P} \to \boldsymbol{Q}$, which satisfy the *fundamental property*:

  $$\mathit{left}(\boldsymbol{f})(p) \leq_{\boldsymbol{Q}} q \text{ iff } p \leq_{\boldsymbol{P}} \mathit{right}(\boldsymbol{f})(q)$$

  for all elements $q \in \boldsymbol{Q}$ and $p \in \boldsymbol{P}$.

  – An adjoint pair $\boldsymbol{f} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ is an adjunction $\langle \mathit{left}(\boldsymbol{f}), \mathit{right}(\boldsymbol{f}), \eta, \varepsilon \rangle : \boldsymbol{P} \to \boldsymbol{Q}$, where the preorders are consider categories, the unit $\eta : Id_P \Rightarrow \mathit{left}(\boldsymbol{f}) \cdot \mathit{right}(\boldsymbol{f})$ corresponds to the induced closure operator on $\boldsymbol{P}$, and the counit $\varepsilon : \mathit{right}(\boldsymbol{f}) \cdot \mathit{left}(\boldsymbol{f}) \Rightarrow Id_Q$ corresponds to the induced interior operator on $\boldsymbol{Q}$.

  – An adjoint pair $\boldsymbol{f} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ is an infomorphism $\boldsymbol{f} : \boldsymbol{Q} \rightleftarrows \boldsymbol{P}$ from classification $\boldsymbol{Q} = \langle Q, Q, \leq_{\boldsymbol{Q}} \rangle$ to classification $\boldsymbol{P} = \langle P, P, \leq_{\boldsymbol{P}} \rangle$, where the preorders are regarded as classifications.

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

(2) (CNG$function source)
    (CNG$signature source adjoint-pair ORD$preorder)

(3) (CNG$function target)
    (CNG$signature target adjoint-pair ORD$preorder)

(4) (CNG$function left)
    (CNG$signature left adjoint-pair ORD.FTN$monotonic-function)
    (forall (?a (adjoint-pair ?a))
        (and (= (ORD.FTN$source (left ?a)) (source ?a))
             (= (ORD.FTN$target (left ?a)) (target ?a))))

(5) (CNG$function right)
    (CNG$signature right adjoint-pair ORD.FTN$monotonic-function)
    (forall (?a (adjoint-pair ?a))
        (and (= (ORD.FTN$source (right ?a)) (target ?a))
             (= (ORD.FTN$target (right ?a)) (source ?a))))

    (forall (?a (adjoint-pair ?a)
             ?p ((ORD$class (source ?a)) ?p)
             ?q ((ORD$class (target ?a)) ?q))
        (<=> ((ORD$extent (target ?a)) [((left ?a) ?p) ?q])
             ((ORD$extent (source ?a)) [?p ((right ?a) ?q)])))
```

- Any adjoint pair is an infomorphism. The CNG function in axiom (6) realizes this assertion.

```
(6) (CNG$function infomorphism)
    (CNG$signature infomorphism
        adjoint-pair CLS.INFO$infomorphism)
    (forall (?a (adjoint-pair ?a))
        (and (= (CLS.INFO$source (infomorphism ?a))
                (ORD$classification (target ?a)))
             (= (CLS.INFO$target (infomorphism ?a))
                (ORD$classification (source ?a)))
             (= (instance (classification ?a))
                (left ?a))
             (= (type (classification ?a))
                (right ?a)))))
```

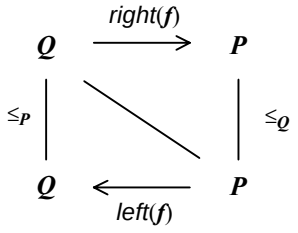


Figure 2: Adjoint Pair as an Infomorphism

- The fundamental property of an adjoint pair $\boldsymbol{f} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ expresses the bonding classification of a bond $\mathit{bond}(\boldsymbol{f}) : \mathit{cls}(\boldsymbol{B}) \rightleftarrows \mathit{cls}(\boldsymbol{A})$ associated with the adjoint pair. Although this could be defined "from scratch," here it is defined in terms of the bond of the functional infomorphism associated with $\boldsymbol{f}$.

```
(7) (CNG$function bond)
    (CNG$signature bond adjoint-pair CLS.BND$bond)
    (forall (?a (adjoint-pair ?a))
```

```
        (and (= (CLS.BND$source (bond ?a)) (ORD$classification (target ?a)))
             (= (CLS.BND$target (bond ?a)) (ORD$classification (source ?a)))
             (= (CLS.BND$classification (bond ?a))
                (CLS.INFO$bond (infomorphism ?a)))))
```

- Duality can be extended to adjoint pairs. For any adjoint pair $\boldsymbol{f} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$, the *opposite* or *dual* of $\boldsymbol{f}$ is the adjoint pair $\boldsymbol{f}^{\perp} : \boldsymbol{Q}^{\perp} \rightleftarrows \boldsymbol{P}^{\perp}$, whose source preorder is the opposite of the target of $\boldsymbol{f}$, whose target preorder is the opposite of the source of $\boldsymbol{f}$, whose left monotonic function is the right of $\boldsymbol{f}$, and whose right monotonic function is the left of $\boldsymbol{f}$.

  Axiom (8) specifies the opposite operator on adjoint pairs.

```
(8) (CNG$function opposite)
    (CNG$signature opposite adjoint-pair adjoint-pair)
    (forall (?f (adjoint-pair ?f))
        (and (= (source (opposite ?f)) (ORD$opposite (target ?f)))
             (= (target (opposite ?f)) (ORD$opposite (source ?f)))
             (= (left (opposite ?f)) (ORD.FTN$opposite (right ?f)))
             (= (right (opposite ?f)) (ORD.FTN$opposite (left ?f)))))
```

- Two adjoint pairs $\boldsymbol{f} : \boldsymbol{O} \rightleftarrows \boldsymbol{P}$ and $\boldsymbol{g} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}$ are *composable* when the target order of $\boldsymbol{f}$ is the same as the source order of $\boldsymbol{g}$. There is a binary CNG function *composition*, which takes two composable adjoint pairs and returns their composition.

```
(10) (CNG$function composition)
     (CNG$signature composition adjoint-pair adjoint-pair adjoint-pair)
      (forall (?f (adjoint-pair ?f) ?g (adjoint-pair ?g))
          (<=> (exists (?h (adjoint-pair ?h)) (= (composition ?f ?g) ?h))
               (= (target ?f) (source ?g)))))
      (forall (?f (adjoint-pair ?f) ?g (adjoint-pair ?g))
          (=> (= (target ?f) (source ?g))
              (and (= (source (composition ?f ?g)) (source ?f))
                   (= (target (composition ?f ?g)) (target ?g))
                   (= (left (composition ?f ?g))
                      (ORD.FTN$composition (left ?f) (left ?g)))
                   (= (right (composition ?f ?g))
                      (ORD.FTN$composition (right ?g) (right ?f)))))))
```

- For any preorder $\boldsymbol{P}$ there is an identity adjoint pair $\mathit{identity}_{\boldsymbol{P}}$.

```
(11) (CNG$function identity)
     (CNG$signature identity ORD$preorder adjoint-pair)
     (forall (?o (ORD$preorder ?o))
         (and (= (source (identity ?o)) ?o)
              (= (target (identity ?o)) ?o)
              (= (left (identity ?o)) (ORD.FTN$identity ?o))
              (= (right (identity ?o)) (ORD.FTN$identity ?o))))
```

Galois Connections

ORD.GC

- A *Galois connection* is an adjoint pair $\boldsymbol{f} : \boldsymbol{P} \rightleftarrows \boldsymbol{Q}^{\mathrm{op}}$ from preorder $\boldsymbol{P}$ to the opposite of preorder $\boldsymbol{Q}$; that is, it is a pair $\boldsymbol{f} = \langle \mathit{right}(\boldsymbol{f}), \mathit{left}(\boldsymbol{f}) \rangle$ of oppositely directed monotonic functions, $\mathit{right}(\boldsymbol{f}) : \boldsymbol{Q}^{\mathrm{op}} \to \boldsymbol{P}$ and $\mathit{left}(\boldsymbol{f}) : \boldsymbol{P} \to \boldsymbol{Q}^{\mathrm{op}}$, which satisfy the *fundamental property*:

  $$\mathit{left}(\boldsymbol{f})(p) \geq_{\boldsymbol{Q}} q \text{ iff } p \leq_{\boldsymbol{P}} \mathit{right}(\boldsymbol{f})(q)$$

  or

  $$\mathit{right}(\boldsymbol{f})(q) \geq_{\boldsymbol{P}} p \text{ iff } q \leq_{\boldsymbol{Q}} \mathit{left}(\boldsymbol{f})(p)$$

  for all elements $q \in \boldsymbol{Q}$ and $p \in \boldsymbol{P}$.

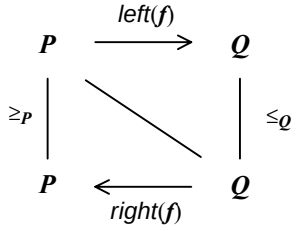


Figure 3: Galois Connection

```
(1) (CNG$conglomerate galois-connection)

(2) (CNG$function source)
```

```
        (CNG$signature source galois-connection preorder)

    (3) (CNG$function target)
        (CNG$signature target galois-connection preorder)

    (4) (CNG$function left)
        (CNG$signature left galois-connection monotonic-function)
        (forall (?a (galois-connection ?a))
            (and (= (ORD.FTN$source (left ?a)) (source ?a))
                 (= (ORD.FTN$target (left ?a)) (opposite (target ?a)))))

    (5) (CNG$function right)
        (CNG$signature right galois-connection monotonic-function)
        (forall (?a (galois-connection ?a))
            (and (= (ORD.FTN$source (right ?a)) (opposite (target ?a)))
                (= (ORD.FTN$target (right ?a)) (source ?a))))

        (forall (?a (galois-connection ?a)
                 ?p ((ORD$class (source ?a)) ?p)
                 ?q ((ORD$class (target ?a)) ?q))
            (<=> ((ORD$extent (target ?a)) [?q ((left ?a) ?p)])
                 ((ORD$extent (source ?a)) [?p ((right ?a) ?q)])))
```

○ The following properties can be proven.

```
    (forall (?a (galois-connection ?a)
             ?p ((ORD$class (source ?a)) ?p))
        (and ((ORD$extent (source ?a)) [?p ((right ?a) ((left ?a) ?p))])
             (= ?p ((left ?a) ((right ?a) ((left ?a) ?p))))))

    (forall (?a (galois-connection ?a)
             ?q ((ORD$class (target ?a)) ?q))
        (and ((ORD$extent (target ?a)) [?q ((left ?a) ((right ?a) ?q))])
             (= ?q ((right ?a) ((left ?a) ((right ?a) ?q))))))
```

○ Using these properties, we can show that there are two closure operators for any Galois connection.

```
(6) (CNG$function source-closure)
    (CNG$signature source-closure galois-connection ORD.CLSR$closure-operator)
    (forall (?a (galois-connection ?a))
        (and (= (ORD.CLSR$class (source-closure ?a))
                (ORD$class (source ?a)))
             (= (ORD.CLSR$function (source-closure ?a))
                (SET.FTN$composition
                    (ORD.FTN$function (left ?a))
                    (ORD.FTN$function (right ?a))))))

(7) (CNG$function target-closure)
    (CNG$signature target-closure galois-connection ORD.CLSR$closure-operator)
    (forall (?a (galois-connection ?a))
        (and (= (ORD.CLSR$class (target-closure ?a))
                (ORD$class (target ?a)))
             (= (ORD.CLSR$function (target-closure ?a))
                (SET.FTN$composition
                    (ORD.FTN$function (right ?a))
                    (ORD.FTN$function (left ?a))))))
```

The Namespace of Large Concept Lattices

This namespace will represent large concept lattices and their morphisms. Some of the terms introduced in this namespace are listed in Table 1.

Table 1: Terms introduced into the large concept lattice namespace

	Conglomerate	Function
CL	'concept-lattice'	'**complete-lattice**', 'instance', 'type' 'instance-embedding', 'type-embedding' '**classification**', 'opposite', 'instance-power'
CL .MOR	'concept-morphism'	'source', 'target', '**adjoint-pair**', 'instance', 'type' '**infomorphism**' 'instance-join', 'type-meet', 'right-representation', 'extent', 'intent', 'left-representation' 'representation' 'opposite', 'composition', 'identity' 'instance-power'

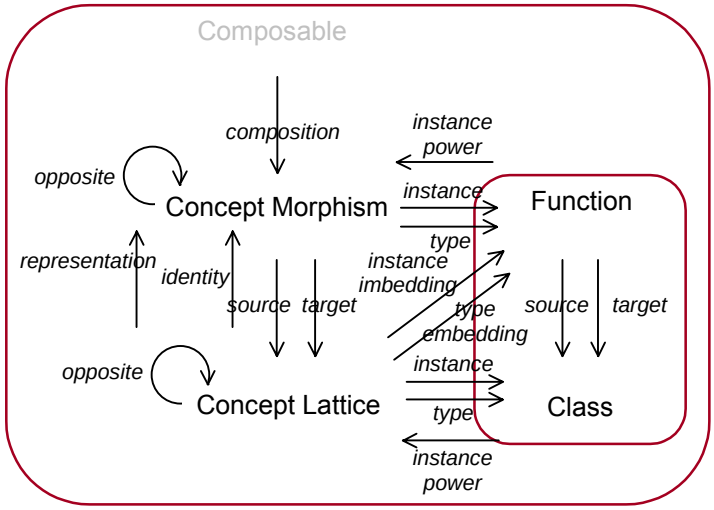


Diagram 3: Core Conglomerates and Functions for Concept Lattices

Concept Lattices

CL

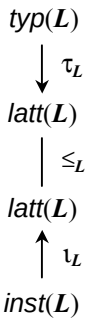


Figure 4: Concept Lattice

- An (abstract) *concept lattice* $\boldsymbol{L} = \langle \mathit{latt}(\boldsymbol{L}), \mathit{inst}(\boldsymbol{L}), \mathit{typ}(\boldsymbol{L}), \iota_{\boldsymbol{L}}, \tau_{\boldsymbol{L}} \rangle$ consists of a complete lattice $\mathit{latt}(\boldsymbol{L})$, two classes $\mathit{inst}(\boldsymbol{L})$ and $\mathit{typ}(\boldsymbol{L})$ called the instance class and the type class of $\boldsymbol{L}$, respectively; along with two functions, an instance embedding function $\iota_{\boldsymbol{L}} : \mathit{inst}(\boldsymbol{L}) \to \mathit{latt}(\boldsymbol{L})$ and a type embedding function $\tau_{\boldsymbol{L}} : \mathit{typ}(\boldsymbol{L}) \to \mathit{latt}(\boldsymbol{L})$, which satisfying the following conditions.
  - The image $\iota_{\boldsymbol{L}}(\mathit{inst}(\boldsymbol{A}))$ is join-dense in $\mathit{latt}(\boldsymbol{L})$.
  - The image $\tau_{\boldsymbol{L}}(\mathit{typ}(\boldsymbol{A}))$ is meet-dense in $\mathit{latt}(\boldsymbol{L})$.

```
(1) (CNG$conglomerate concept-lattice)

(2) (CNG$function complete-lattice)
    (CNG$signature lattice concept-lattice LAT$complete-lattice)

(3) (CNG$function instance)
    (CNG$signature instance concept-lattice SET$class)

(4) (CNG$function type)
    (CNG$signature type concept-lattice SET$class)

(5) (CNG$function instance-embedding)
    (CNG$signature instance-embedding concept-lattice SET.FTN$function)
    (forall (?l (concept-lattice ?l))
        (and (= (SET.FTN$source (instance-embedding ?l))
                (instance ?l))
             (= (SET.FTN$target (instance-embedding ?l))
                (ORD$class (LAT$underlying (complete-lattice ?l))))))

(5) (CNG$function type-embedding)
    (CNG$signature type-embedding concept-lattice SET.FTN$function)
    (forall (?l (concept-lattice ?l))
        (and (= (SET.FTN$source (type-embedding ?l))
                (type ?l))
             (= (SET.FTN$target (type-embedding ?l))
                (ORD$class (LAT$underlying (complete-lattice ?l))))))

    (forall (?a (CLS$classification ?a))
        (and ((join-dense (complete-lattice ?l))
                 (SET.FTN$image (instance-embedding ?l)))
             ((meet-dense (complete-lattice ?l))
                 (SET.FTN$image (type-embedding ?l)))))
```

- Any concept lattice $\boldsymbol{L} = \langle \mathit{latt}(\boldsymbol{L}), \mathit{inst}(\boldsymbol{L}), \mathit{typ}(\boldsymbol{L}), \iota_{\boldsymbol{L}}, \tau_{\boldsymbol{L}} \rangle$ has an associated classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$ whose incidence relation is defined by $i \models_{\boldsymbol{A}} t$ iff $\iota(i) \leq_{\boldsymbol{A}} \tau(t)$.

```
(6) (CNG$function classification)
    (CNG$signature classification concept-lattice CLS$classification)
    (forall (?l (concept-lattice ?l))
        (and (= (CLS$instance (classification ?l)) (instance ?l))
             (= (CLS$type (classification ?l)) (type ?l))
             (forall (?i ((instance ?l) ?i) ?t ((type ?l) ?t))
                 (<=> ((CLS$incidence (classification ?l)) [?i ?t])
                      ((ORD$extent (LAT$underlying (complete-lattice ?l)))
                          [(instance-embedding ?l) ?i) (type-embedding ?l) ?t)])))))
```

- From properties discussed above, it can be immediately proven that the composition of '`concept-lattice`' and '`classification`' is the identity on the '`classification`' conglomerate. We state this in an external namespace.

```
    (forall (?c (CLS$classification ?c))
        (= (CL$classification (CLS.CL$concept-lattice ?c)) ?c))
```

- Also, from properties discussed above, it can be immediately proven that for any complete lattice L, the complete lattice of the concept lattice of L is L itself; that is, that the composition of '`concept-lattice`' and '`complete-lattice`' is the identity on the '`complete-lattice`' conglomerate. We state this in an external namespace.

```
(forall (?l (LAT$complete-lattice ?l))
    (= (CL$complete-lattice (LAT$concept-lattice ?l)) ?l))
```

- For any concept lattice $\boldsymbol{L} = \langle \mathit{latt}(\boldsymbol{L}), \mathit{inst}(\boldsymbol{L}), \mathit{typ}(\boldsymbol{L}), \iota_{\boldsymbol{L}}, \tau_{\boldsymbol{L}} \rangle$, the *opposite* or *dual* of $\boldsymbol{L}$ is the concept lattice $\boldsymbol{L}^{\perp} = \langle \mathit{latt}(\boldsymbol{L})^{\perp}, \mathit{typ}(\boldsymbol{L}), \mathit{inst}(\boldsymbol{L}), \tau_{\boldsymbol{L}}, \iota_{\boldsymbol{L}} \rangle$, whose instances are the types of $\boldsymbol{L}$, whose types are the instances of $\boldsymbol{L}$, whose instance embedding function is the type embedding function of $\boldsymbol{L}$, whose type embedding function is the instance embedding function of $\boldsymbol{L}$, and whose complete lattice is the opposite of the complete lattice of $\boldsymbol{L}$ (turn it upside down). Axiom (7) specifies the opposite operator on concept lattices.

```
(7) (CNG$function opposite)
    (CNG$signature opposite concept-lattice concept-lattice)
    (forall (?l (concept-lattice ?l))
        (and (= (complete-lattice (opposite ?l))
                (LAT$opposite (complete-lattice ?l)))
             (= (instance (opposite ?l)) (type ?l))
             (= (type (opposite ?l)) (instance ?l))
             (= (instance-embedding (opposite ?l)) (type-embedding ?l))
             (= (type-embedding (opposite ?l)) (instance-embedding ?l))))
```

- For any class A the *instance power concept lattice* $\wp A = \langle\langle \wp A, \subseteq_A, \cap_A, \cup_A \rangle, A, \wp A, \{\text{-}\}_A, id_{\wp A} \rangle$ over A is defined as follows: the complete lattice is the power lattice generated by A, the instance class is A; the type class is the power class $\wp A$ (so that a type is a subclass of A), the instance embedding function is the singleton function for A, and the type-embedding function is the identity. Axiom (8) specifies the instance power operator from classes to concept lattices.

```
(8) (CNG$function instance-power)
    (CNG$signature instance-power SET$class concept-lattice)
    (forall (?c (SET$class ?c))
        (and (= (complete-lattice (instance-power ?c)) (LAT$power ?c))
             (= (instance (instance-power ?c)) ?c)
             (= (type (instance-power ?c)) (SET$power ?c))
             (= (instance-embedding (instance-power ?c)) (SET.FTN$singleton ?c))
             (= (type-embedding (instance-power ?c))
                (SET.FTN$identity (SET$power ?c)))))
```

Concept Morphisms

`CL.MOR`

- Concept lattices are related through concept morphisms. An (abstract) *concept morphism* $\boldsymbol{f} : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$ from (abstract) concept lattice $\boldsymbol{L}$ to (abstract) concept lattice $\boldsymbol{K}$ consists of a pair of ordinary oppositely directed functions, $\mathit{inst}(\boldsymbol{f}) : \mathit{inst}(\boldsymbol{K}) \to \mathit{inst}(\boldsymbol{L})$ and $\mathit{typ}(\boldsymbol{f}) : \mathit{typ}(\boldsymbol{L}) \to \mathit{typ}(\boldsymbol{K})$, between instance classes and type classes, and an adjoint pair $\mathit{adj}(\boldsymbol{f}) : \boldsymbol{K} \rightleftarrows \boldsymbol{L}$ of monotonic functions, where the right adjoint $\mathit{right}(\boldsymbol{f}) : \boldsymbol{L} \to \boldsymbol{K}$ is a monotonic function in the forward direction (for the concept lattice morphism, not the adjoint pair) that preserves types (in the sense that the upper rectangle in Figure 5 is commutative)

$$\tau_{\boldsymbol{L}} \cdot \mathit{right}(\mathit{adj}(\boldsymbol{f})) = \mathit{typ}(\boldsymbol{f}) \cdot \tau_{\boldsymbol{K}}$$

and the left adjoint $\mathit{left}(\boldsymbol{f}) : \boldsymbol{K} \to \boldsymbol{L}$ is a monotonic function in the reverse direction that preserves instances (in the sense that the lower rectangle in Figure 5 is commutative)

$$\iota_{\boldsymbol{K}} \cdot \mathit{left}(\mathit{adj}(\boldsymbol{f})) = \mathit{inst}(\boldsymbol{f}) \cdot \iota_{\boldsymbol{L}} \ .$$

Axiom (1) specifies the conglomerate of concept morphisms. Axiom (4) defines the overlying adjoint pair. Note the contravariance between the

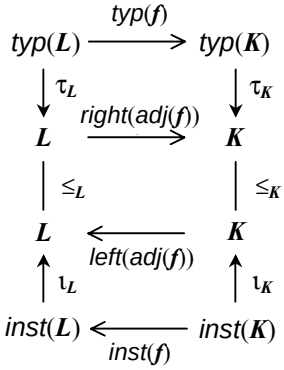


Figure 5: Concept Lattice Morphism

concept morphism and the adjoint pair – the concept morphism is oriented in the same direction as the type function, whereas the adjoint pair is oriented in the same direction as the left adjoint.

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

(2) (CNG$function source)
    (CNG$signature source concept-morphism CL$concept-lattice)

(3) (CNG$function target)
    (CNG$signature target concept-morphism CL$concept-lattice)

(4) (CNG$function adjoint-pair)
    (CNG$signature adjoint-pair concept-morphism LAT.ADJ$adjoint-pair)
    (forall (?f (concept-morphism ?f))
        (and (= (LAT.ADJ$source (adjoint-pair ?f))
                (CL$complete-lattice (target ?f)))
             (= (LAT.ADJ$target (adjoint-pair ?f))
                (CL$complete-lattice (source ?f)))))

(5) (CNG$function instance)
    (CNG$signature instance concept-morphism SET.FTN$function)
    (forall (?f (concept-morphism ?f))
        (and (= (SET.FTN$source (instance ?f)) (CL$instance (target ?f)))
             (= (SET.FTN$target (instance ?f)) (CL$instance (source ?f)))
             (= (SET.FTN$composition
                    (CL$instance-embedding (target ?f))
                    (LAT.ADJ$left (adjoint-pair ?f)))
                (SET.FTN$composition
                    (instance ?f)
                    (CL$instance-embedding (source ?f))))))

(6) (CNG$function type)
    (CNG$signature type concept-morphism SET.FTN$function)
    (forall (?f (concept-morphism ?f))
        (and (= (SET.FTN$source (type ?f)) (CL$type (source ?f)))
             (= (SET.FTN$target (type ?f)) (CL$type (target ?f)))
             (= (SET.FTN$composition
                    (CL$type-embedding (source ?f))
                    (LAT.ADJ$right (adjoint-pair ?f)))
                (SET.FTN$composition
                    (type ?f)
                    (CL$type-embedding (target ?f))))))
```

- Any concept morphism $\boldsymbol{f} : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$ has an associated infomorphism $\mathit{info}(\boldsymbol{f}) : \mathit{cls}(\boldsymbol{L}) \rightleftarrows \mathit{cls}(\boldsymbol{K})$ whose fundamental property, expressed as

  $$\mathit{inst}(\mathit{info}(\boldsymbol{f}))(i) \vDash_{\mathit{cls}(\boldsymbol{L})} t \text{ iff } i \vDash_{\mathit{cls}(\boldsymbol{K})} \mathit{typ}(\mathit{info}(\boldsymbol{f}))(t)$$

  for all instances $i \in \mathit{inst}(\boldsymbol{K})$ and all types $t \in \mathit{typ}(\boldsymbol{L})$, is an easy translation of the adjointness condition for the adjoint pair $\mathit{adj}(\boldsymbol{f})$ and the commutativity of the instance/type functions with the left/right monotonic functions.

```
(7) (CNG$function infomorphism)
    (CNG$signature infomorphism concept-morphism CLS.INFO$infomorphism)
    (forall (?f (concept-morphism ?l))
        (and (= (CLS.INFO$instance (infomorphism ?f)) (instance ?f))
             (= (CLS.INFO$type (infomorphism ?f)) (type ?f))))
```

- From properties discussed above, it can be immediately proven that the composition of '`concept-morphism`' and '`infomorphism`' is the identity on the '`infomorphism`' conglomerate. We state this in an external namespace.

```
    (forall (?f (CLS.INFO$infomorphism ?f))
        (= (CL.MOR$infomorphism (CLS.CL$concept-morphism ?f)) ?f))
```

- For any concept lattice $\boldsymbol{L}$, the concept lattice of the classification of $\boldsymbol{L}$ is related to $\boldsymbol{L}$ through the following two functions.

```
(8) (CNG$function instance-join)
    (CNG$signature instance-join CL$concept-lattice SET.FTN$function)
```

```
        (forall (?l (CL$concept-lattice ?l))
            (and (= (source (instance-join ?l))
                    (ORD$class (LAT$underlying (CL$complete-lattice
                        (CLS$concept-lattice (CL$classification ?l))))))
                 (= (target (instance-join ?l))
                    (ORD$class (LAT$underlying (CL$complete-lattice ?l))))
                 (= (instance-join ?l)
                    (SET.FTN$composition
                        (SET.FTN$composition
                            (CLS$extent (CL$classification ?l))
                            (SET.FTN$power (CL$instance-embedding ?l)))
                        (LAT$join (CL$complete-lattice ?l)))))))

    (9) (CNG$function type-meet)
        (CNG$signature type-meet CL$concept-lattice SET.FTN$function)
        (forall (?l (CL$concept-lattice ?l))
            (and (= (source (type-meet ?l))
                    (ORD$class (LAT$underlying (CL$complete-lattice
                        (CLS$concept-lattice (CL$classification ?l))))))
                 (= (target (type-meet ?l))
                    (ORD$class (LAT$underlying (CL$complete-lattice ?l))))
                 (= (type-meet ?l)
                    (SET.FTN$composition
                        (SET.FTN$composition
                            (CLS$intent (CL$classification ?l))
                            (SET.FTN$power (CL$type-embedding ?l)))
                        (LAT$meet (CL$complete-lattice ?l)))))))
```

- The previous two functions can be shown to be identical monotonic functions.

```
(forall (?l (CL$concept-lattice ?l))
    (and (= (instance-join ?l) (type-meet ?l)))
         (exists (?f (ORD.FTN$monotonic-function ?))
             (= (instance-join ?l) (ORD.FTN$function ?f)))))
```

- Let us call this common function the *right representation* of ***L***.

```
(10) (CNG$function right-representation)
     (CNG$signature right-representation CL$concept-lattice SET.FTN$function)
     (= right-representation instance-join)
```

- Any concept lattice ***L*** is indirectly related to the concept lattice of the classification of ***L*** through the following two functions. The *extent* of a concept in ***L*** is the class of all instances whose generated concept is at or below the concept. The *intent* is the dual notion. As we shall observe and axiomatize, both the extent and intent represent concepts of ***L***.

```
(11) (CNG$function extent)
     (CNG$signature extent CL$concept-lattice SET.FTN$function)
     (forall (?l (CL$concept-lattice ?l))
         (and (= (source (extent ?l))
                 (ORD$class (LAT$underlying (CL$complete-lattice ?l))))
              (= (target (extent ?l))
                 (SET$power (CL$instance ?l)))
              (= (extent ?l)
              (= (SET.FTN$composition (extent ?l) (CLS$extent (CL$classification ?l)))
                 (SET.FTN$composition
                     (SET.FTN$singleton
                         (ORD$class (LAT$underlying (CL$complete-lattice ?l))))
                     (SET.FTN$composition
                         ((ORD$down (LAT$underlying (CL$complete-lattice ?l)))
                         (SET.FTN$inverse-image (CL$instance-embedding ?l)))))))))

(12) (CNG$function intent)
     (CNG$signature intent CL$concept-lattice SET.FTN$function)
     (forall (?l (CL$concept-lattice ?l))
         (and (= (source (intent ?l))
                 (ORD$class (LAT$underlying (CL$complete-lattice ?l))))
              (= (target (intent ?l))
                 (SET$power (CL$type ?l)))
              (= (intent ?l)
                 (SET.FTN$composition
```

```
                    (SET.FTN$singleton
                        (ORD$class (LAT$underlying (CL$complete-lattice ?l))))
                    (SET.FTN$composition
                        ((ORD$up (LAT$underlying (CL$complete-lattice ?l)))
                        (SET.FTN$inverse-image (CL$type-embedding ?l)))))))
```

- The following fact can be proven: there is a unique function, whose source class is the source of the extent and intent functions, whose target is the class underlying the concept lattice of the classification of ***L***, whose composition with the extent function of the classification of ***L*** is the above extent function, and whose composition with the intent function of the classification of ***L*** is the above intent function. Let us call this function the *left representation* of ***L***.

```
(13) (CNG$function left-representation)
     (CNG$signature left-representation CL$concept-lattice SET.FTN$function)
     (forall (?l (CL$concept-lattice ?l))
         (and (= (source (left-representation ?l))
                 (ORD$class (LAT$underlying (CL$complete-lattice ?l))))
              (= (target (left-representation ?l))
                 (ORD$class (LAT$underlying (CL$complete-lattice
                     (CLS$concept-lattice (CL$classification ?l))))))
              (= (SET.FTN$composition
                     (left-representation ?l)
                     (CLS$extent (CL$classification ?l)))
                 (extent ?l))
              (= (SET.FTN$composition
                     (left-representation ?l)
                     (CLS$intent (CL$classification ?l)))
                 (intent ?l)))))
```

- For any concept lattice ***L***, it can be proven that the left and right representation functions are inverse to each other. This demonstrates that the concept lattice of the classification of ***L*** represents ***L*** via left representation (extent and intent functions).

```
     (forall (?l (CL$concept-lattice ?l))
         (and (= (SET.FTN$composition
                     (left-representation ?l)
                     (right-representation ?l))
                 (SET.FTN$identity
                     (ORD$class (LAT$underlying (CL$complete-lattice ?l)))))
              (= (SET.FTN$composition
                     (right-representation ?l)
                     (left-representation ?l))
                 (SET.FTN$identity
                     (ORD$class (LAT$underlying (CL$complete-lattice
                         (CLS$concept-lattice (CL$classification ?l)))))))))
```

- We rephrase this in terms of concept morphisms: for any concept lattice ***L***, there is a *representation* concept morphism from ***L*** to the concept lattice of the classification of ***L***. This is the $\boldsymbol{L}^{\text{th}}$ component of a natural isomorphism, demonstrating that the following quasi-categories are categorically equivalent:

  Classification ≡ Concept Lattice.

```
(14) (CNG$function representation)
     (CNG$signature representation CL$concept-lattice concept-morphism)
     (forall (?l (CL$concept-lattice ?l))
         (and (= (source (representation ?l)) ?l)
              (= (target (representation ?l))
                 (CLS$concept-lattice (CL$classification ?l)))
              (= (LAT.ADJ$left (adjoint-pair (representation ?l)))
                 (left-representation ?l))
              (= (LAT.ADJ$right (adjoint-pair (representation ?l)))
                 (right-representation ?l))
              (= (instance (representation ?l))
                 (SET.FTN$identity (CL$instance ?l)))
              (= (type (representation ?l))
                 (SET.FTN$identity (CL$instance ?l))))))
```

- In addition, from properties discussed above, it can be immediately proven that for any (complete lattice) adjoint pair *f*, the adjoint pair of the concept morphism of *f* is *f* itself; that is, that the

composition of '`ORD.LAT$concept-morphism`' and '`adjoint-pair`' is the identity on the '`adjoint-pair`' conglomerate. We state this in an external namespace.

```
(forall (?f (LAT.ADJ$adjoint-pair ?f))
    (= (CL.MOR$adjoint-pair (LAT.ADJ$concept-morphism ?f)) ?f))
```

Duality can be extended to concept morphisms. For any concept morphism $\boldsymbol{f} : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$, the *opposite* or *dual* of $\boldsymbol{f}$ is the concept morphism $\boldsymbol{f}^{\perp} : \boldsymbol{K}^{\perp} \rightleftarrows \boldsymbol{L}^{\perp}$, whose source concept lattice is the opposite of the target of $\boldsymbol{f}$, whose target concept lattice is the opposite of the source of $\boldsymbol{f}$, whose adjoint pair is the opposite of the adjoint pair of $\boldsymbol{f}$, whose instance function is the type function of $\boldsymbol{f}$, whose type function is the instance function of $\boldsymbol{f}$, and whose preservation conditions have been dualized.

Axiom (15) specifies the opposite operator on concept morphisms.

```
(15) (CNG$function opposite)
     (CNG$signature opposite concept-morphism concept-morphism)
     (forall (?f (infomorphism ?f))
         (and (= (source (opposite ?f)) (CL$opposite (target ?f)))
              (= (target (opposite ?f)) (CL$opposite (source ?f)))
              (= (adjoint-pair (opposite ?f)) (LAT.ADJ$opposite (adjoint-pair ?f)))
              (= (instance (opposite ?f)) (type ?f))
              (= (type (opposite ?f)) (instance ?f))))
```

- The function '`composition`' operates on any two concept morphisms that are composable in the sense that the target concept lattice of the first is equal to the source concept lattice of the second. Composition, defined in axiom (16), produces a concept morphism, whose components are constructed using composition.

```
(16) (CNG$function composition)
     (CNG$signature composition concept-morphism concept-morphism concept-morphism)
     (forall (?f (concept-morphism ?f) ?g (concept-morphism ?g))
         (<=> (exists (?h (concept-morphism ?h)) (= (composition ?f ?g) ?h))
              (= (target ?f) (source ?g))))
     (forall (?f (concept-morphism ?f) ?g (concept-morphism ?g))
         (=> (= (target ?f) (source ?g))
             (and (= (source (composition ?f ?g)) (source ?f))
                  (= (target (composition ?f ?g)) (target ?g))
                  (= (adjoint-pair (composition ?f ?g))
                     (CL.MOR$composition (adjoint-pair ?g) (adjoint-pair ?f)))
                  (= (instance (composition ?f ?g))
                     (SET.FTN$composition (instance ?g) (instance ?f)))
                  (= (type (composition ?f ?g))
                     (SET.FTN$composition (type ?f) (type ?g))))))
```

- The function '`identity`' defined in axiom (17) associates a well-defined (identity) concept morphism with any concept lattice, whose components are identities.

```
(17) (CNG$function identity)
     (CNG$signature identity CL$concept-lattice concept-morphism)
     (forall (?l (CL$concept-lattice ?l))
         (and (= (source (identity ?l)) ?l)
              (= (target (identity ?l)) ?l)
              (= (adjoint-pair (identity ?l))
                 (CL.MOR$identity (complete-lattice ?l)))
              (= (instance (identity ?l))
                 (SET.FTN$identity (CL$instance ?l)))
              (= (type (identity ?c))
                 (SET.FTN$identity (CL$type ?l)))))
```

- A very useful generic concept morphism represents the "instance power concept morphism construction." For any class function $f : B \to A$ the components of the *instance power concept morphism*

$$\wp f = \langle\langle \wp f, f^{-1} \rangle, f, f^{-1} \rangle : \wp A \rightleftarrows \wp B$$

over f are defined as follows: the source concept lattice is the instance power concept lattice $\wp A = \langle\langle \wp A, \subseteq_A, \cap_A, \cup_A \rangle, A, \wp A, \{\text{-}\}_A, id_{\wp A} \rangle$ over A, the target concept lattice is the instance power concept

lattice $\wp B = \langle\langle \wp\wp B, \subseteq_B, \cap_B, \cup_B \rangle, B, \wp B, \{\text{-}\}_B, id_{\wp B} \rangle$ over B, the adjoint pair is the (complete lattice) power adjoint pair over f, the instance function is f, and the type function is the inverse image function $f^{-1} : \wp A \to \wp B$ from the power-class of A to the power-class of B. Note the contravariance.

```
(18) (CNG$function instance-power)
     (CNG$signature instance-power SET.FTN$function concept-morphism)
     (forall (?f (SET.FTN$function ?f))
         (and (= (source (instance-power ?f)) (CL$instance-power (SET.FTN$target ?f)))
              (= (target (instance-power ?f)) (CL$instance-power (SET.FTN$source ?f)))
              (= (adjoint-pair (instance-power ?f)) (LAT.ADJ$power ?f))
              (= (instance (instance-power ?f)) ?f)
              (= (type (instance-power ?f)) (SET.FTN$inverse-image ?f))))
```

The Namespace of Large Complete Lattices

This namespace will represent large complete lattices and their adjoint pairs. Some of the terms introduced in this namespace are listed in Table 1.

Table 1: Terms introduced into the large complete lattice namespace

	Conglomerate	Function	Example
LAT	'complete-lattice'	'underlying', 'meet', 'join' '**classification**' 'cut' 'opposite', 'power' '**concept-lattice**'	
LAT .ADJ	'adjoint-pair'	'source', 'target', 'underlying', 'left', 'right' '**infomorphism**' '**bond**' '**cut-forward**', '**cut-reverse**' 'opposite', 'composition', 'identity' 'power' '**concept-morphism**'	
LAT .MOR	'homomorphism'	'source', 'target', 'function', 'forward', 'reverse' '**bonding-pair**', '**cut**'	

Complete Lattices

LAT

- A partial order $\boldsymbol{L}$ is a *complete lattice* when the meet and join exist for all classes $S \subseteq L$. Then we use the notation $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$. The underlying partial order is represented by $|\boldsymbol{L}| = \langle L, \leq_L \rangle$.

```
(1) (CNG$conglomerate complete-lattice)

(2) (CNG$function underlying)
    (CNG$signature underlying complete-lattice ORD$partial-order)

(3) (CNG$function meet)
    (CNG$signature meet complete-lattice SET.FTN$function)
    (forall (?l (complete-lattice ?l))
        (and (= (SET.FTN$source (meet ?l)) (SET$power (ORD$class (underlying ?l))))
             (= (SET.FTN$target (meet ?l)) (ORD$class (underlying ?l)))
             (forall (?c ((SET$power (ORD$class (underlying ?l))) ?c))
                 (((greatest ?o) ((lower-bound ?o) ?c)) ((meet ?l) ?c)))))

(4) (CNG$function join)
    (CNG$signature join complete-lattice SET.FTN$function)
    (forall (?l (complete-lattice ?l))
        (and (= (SET.FTN$source (join ?l)) (SET$power (ORD$class (underlying ?l))))
             (= (SET.FTN$target (join ?l)) (ORD$class (underlying ?l)))
             (forall (?c ((SET$power (ORD$class (underlying ?l))) ?c))
                 (((least ?o) ((upper-bound ?o) ?c)) ((join ?l) ?c)))))
```

- Associated with any complete lattice $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$ is the classification $cls(\boldsymbol{L}) = cls(|\boldsymbol{L}|) = \langle L, L, \leq_L \rangle$, which has L-elements as its instances and types, and the lattice order as its incidence.

```
(5) (CNG$function classification)
    (CNG$signature classification complete-lattice CLS$classification)
    (forall (?l (complete-lattice ?l))
        (= (classification ?l)
           (ORD$classification (underlying ?l))))
```

- For any complete lattice $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$, there is a special *cut* function $\mathit{cut}(\boldsymbol{L}) = \updownarrow_L : L \to \mathit{lat}(\mathit{cls}(\boldsymbol{L}))$ defined by $x \mapsto (\downarrow_L x, \uparrow_L x)$, for every element $x \in L$.

```
(6) (CNG$function cut)
    (CNG$signature cut complete-lattice SET.FTN$function)
    (forall (?l (complete-lattice ?l))
        (and (SET.FTN$source (cut ?l)) (ORD$class (underlying ?l)))
             (SET.FTN$target (cut ?l)) (CLS.CL$concept (classification ?l)))
             (= (SET.FTN$composition (cut ?l) (CLS.CL$extent (classification ?l)))
                (ORD$down-embedding (underlying ?l)))
             (= (SET.FTN$composition (cut ?l) (CLS.CL$intent (classification ?l)))
                (ORD$up-embedding (underlying ?l)))))
```

- Here are some preliminary observations that pertain to this cut function.

  On the underlying partial-order of a complete lattice, the composition of the down-embedding and join is the identity on the underlying class. Dually, the composition of the up-embedding and meet is the identity on the underlying class.

```
    (forall (?l (complete-lattice ?l))
        (and (= (SET.FTN$composition (ORD$down-embedding (underlying ?l)) (join ?l))
                (SET.FTN$identity (ORD$class (underlying ?l))))
             (= (SET.FTN$composition (ORD$up-embedding (underlying ?l)) (meet ?l))
                (SET.FTN$identity (ORD$class (underlying ?l))))))
```

- On the associated classification of a complete lattice, the instance generation function factors as the composition of join and the above cut function. Dually, the type generation function is the composition of meet followed by cut.

```
    (forall (?l (complete-lattice ?l))
        (and (= (instance-generation (classification ?l))
                (SET.FTN$composition (join ?l) (cut ?l)))
             (= (type-generation (classification ?l))
                (SET.FTN$composition (meet ?l) (cut ?l)))))
```

- It is important to observe that any concept in $\mathit{lat}(\mathit{cls}(\boldsymbol{L}))$ is of the form $(\downarrow_L x, \uparrow_L x)$ for some element $x \in L$. In fact, the cut function is a bijection, and its inverse function has two expressions – it is the composition of conceptual extent and join, and it is the composition of conceptual intent and meet.

```
(forall (?l (complete-lattice ?l))
    (and (= (SET.FTN$composition
                (cut ?l)
                (SET.FTN$composition (CLS.CL$extent (classification ?l)) (join ?l)))
            (SET.FTN$identity (ORD$class (underlying ?l))))
         (= (SET.FTN$composition
                (SET.FTN$composition (CLS.CL$extent (classification ?l)) (join ?l))
                (cut ?l))
            (SET.FTN$identity (CLS.CL$concept (classification ?l))))
         (= (SET.FTN$composition
                (cut ?l)
                (SET.FTN$composition (CLS.CL$intent (classification ?l)) (meet ?l)))
            (SET.FTN$identity (ORD$class (underlying ?l))))
         (= (SET.FTN$composition
                (SET.FTN$composition (CLS.CL$intent (classification ?l)) (meet ?l))
                (cut ?l))
            (SET.FTN$identity (CLS.CL$concept (classification ?l))))))
```

- For any complete lattice $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$, the *opposite* or *dual* of $\boldsymbol{L}$ is the complete lattice $\boldsymbol{L}^{\perp} = \langle L, \geq_L, \sqcup_L, \sqcap_L \rangle$, whose underlying partial order is the opposite of the underlying partial order of $\boldsymbol{L}$, whose meet is the join of $\boldsymbol{L}$, and whose join is the meet of $\boldsymbol{L}$. Axiom (6) specifies the opposite operator on complete lattices.

```
(6) (CNG$function opposite)
    (CNG$signature opposite complete-lattice complete-lattice)
    (forall (?l (complete-lattice ?l))
        (and (= (underlying (opposite ?l)) (ORD$opposite (underlying ?l)))
             (= (meet (opposite ?l)) (join ?l))
             (= (join (opposite ?l)) (meet ?l)))))
```

- For any class C, the power complete lattice $\wp(C) = \langle \wp C, \subseteq_C, \cap_C, \cup_C \rangle$ is the power class with subclass ordering, intersection as meet and union as join. There is a CNG function *power* : class → complete-lattice that maps a class to its power lattice.

```
(7) (CNG$function power)
    (CNG$signature power SET$class complete-lattice)
    (forall (?c (SET$class ?c))
        (and (= (underlying (power ?c)) (ORD$power ?c))
             (= (meet (power ?c)) (SET$intersection ?c))
             (= (join (power ?c)) (SET$union ?c))))
```

- Any complete lattice $\boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle$ is a concept lattice $\boldsymbol{L} = \langle \boldsymbol{L}, L, L, id_{\boldsymbol{L}}, id_{\boldsymbol{L}} \rangle$, where the instance and type classes are the underlying class of the lattice and the instance and type embeddings are the identity function. Axiom (8) represents this as a CNG function.

```
(8) (CNG$function concept-lattice)
    (CNG$signature concept-lattice complete-lattice CL$concept-lattice)
    (forall (?l (complete-lattice ?l))
        (and (= (CL$complete-lattice (concept-lattice ?l)) ?l)
             (= (CL$instance (concept-lattice ?l)) (ORD$class (underlying ?l)))
             (= (CL$type (concept-lattice ?l)) (ORD$class (underlying ?l)))
             (= (CL$instance-embedding (concept-lattice ?l))
                (SET.FTN$identity (ORD$class (underlying ?l))))
             (= (CL$type-embedding (concept-lattice ?l))
                (SET.FTN$identity (ORD$class (underlying ?l))))))
```

- An easy check shows that the classification of the concept lattice of a complete lattice $\boldsymbol{L}$ is the same as the classification associated with $\boldsymbol{L}$.

```
(forall (?l (complete-lattice ?l))
    (= (CL$classification (concept-lattice ?l))
       (classification ?l)))
```

Complete Adjoint

LAT.ADJ

- Complete lattices are related through *adjoint pairs*. This is a restriction to complete lattices of the adjoint pair notion for preorders. For an adjoint pair $\langle \varphi, \psi \rangle : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$ between complete lattices $\boldsymbol{L} = \langle \boldsymbol{L}, \leq_{\boldsymbol{L}}, \sqcap_{\boldsymbol{L}}, \sqcup_{\boldsymbol{L}} \rangle$, and $\boldsymbol{K} = \langle \boldsymbol{K}, \leq_{\boldsymbol{K}}, \sqcap_{\boldsymbol{K}}, \sqcup_{\boldsymbol{K}} \rangle$, the left adjoint $\varphi : \boldsymbol{L} \to \boldsymbol{K}$ is join-preserving and the right-adjoint $\psi : \boldsymbol{K} \to \boldsymbol{L}$ is meet-preserving. The two functions determine each other as follows.

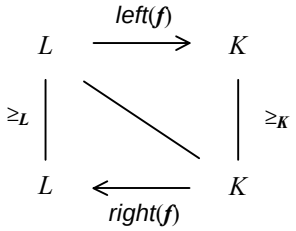


Figure 4: Adjoint Pair

$\varphi(l) = \sqcap_{\boldsymbol{K}} \{k \in K \mid l \leq_{\boldsymbol{L}} \psi(k)\}$

$\psi(k) = \vee_{\boldsymbol{L}} \{l \in L \mid k \leq_{\boldsymbol{K}} \varphi(l)\}$

For example, suppose $\psi : \boldsymbol{K} \to \boldsymbol{L}$ is a meet-preserving monotonic function, and define the function $\varphi : L \to K$ as above.

- If $l_1 \leq_{\boldsymbol{L}} l_2$ then $\{k \in K \mid l_1 \leq_{\boldsymbol{L}} \psi(k)\} \supseteq \{k \in K \mid l_2 \leq_{\boldsymbol{L}} \psi(k)\}$. Hence, $\varphi(l_1) \leq_{\boldsymbol{K}} \varphi(l_2)$.
- If $l \leq_{\boldsymbol{L}} \psi(k)$ then $k \in \{\tilde{k} \in K \mid l \leq_{\boldsymbol{L}} \psi(\tilde{k})\}$. Hence, $\varphi(l) \leq_{\boldsymbol{K}} k$.
- $\psi(\varphi(l)) = \psi(\sqcap_{\boldsymbol{K}} \{k \in K \mid l \leq_{\boldsymbol{L}} \psi(k)\}) = \sqcap_{\boldsymbol{L}} \{\psi(k) \in L \mid l \leq_{\boldsymbol{L}} \psi(k)\} \geq_{\boldsymbol{L}} l$.
- If $\varphi(l) \leq_{\boldsymbol{K}} k$ then $\psi(\varphi(l)) \leq_{\boldsymbol{L}} \psi(k)$. Hence, $l \leq_{\boldsymbol{L}} \psi(k)$.

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

(2) (CNG$function source)
    (CNG$signature source adjoint-pair LAT$complete-lattice)

(3) (CNG$function target)
    (CNG$signature target adjoint-pair LAT$complete-lattice)

(4) (CNG$function underlying)
    (CNG$signature underlying adjoint-pair ORD.ADJ$adjoint-pair)
    (forall (?a (adjoint-pair ?a))
        (and (= (ORD.ADJ$source (underlying ?a)) (LAT$underlying (source ?a)))
             (= (ORD.ADJ$target (underlying ?a)) (LAT$underlying (target ?a)))))
```

- We define the following two terms for convenience of reference.

```
(5) (CNG$function left)
    (CNG$signature left adjoint-pair ORD.FTN$monotonic-function)
    (forall (?a (adjoint-pair ?a))
        (= (left ?a) (ORD.ADJ$left (underlying ?a))))

(6) (CNG$function right)
    (CNG$signature right adjoint-pair ORD.FTN$monotonic-function)
    (forall (?a (adjoint-pair ?a))
        (= (right ?a) (ORD.ADJ$right (underlying ?a))))
```

- Any adjoint pair is an infomorphism. The CNG function in axiom (7) realizes this assertion.

```
(7) (CNG$function infomorphism)
    (CNG$signature infomorphism adjoint-pair CLS.INFO$infomorphism)
    (forall (?a (adjoint-pair ?a))
        (= (infomorphism ?a)
           (ORD.ADJ$infomorphism (underlying ?a))))
```

- Associated with any adjoint pair $\boldsymbol{f} = \langle \mathit{left}(\boldsymbol{f}), \mathit{right}(\boldsymbol{f}) \rangle = \langle \varphi, \psi \rangle : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$, from complete lattice $\boldsymbol{L}$ to complete lattice $\boldsymbol{K}$, is the bond $\mathit{bnd}(\boldsymbol{f}) : \mathit{cls}(\boldsymbol{K}) \rightleftarrows \mathit{cls}(\boldsymbol{L})$ (whose classification relation is) defined by the adjointness property: $l \mathit{bnd}(\boldsymbol{f}) k$ iff $\varphi(l) \leq_{\boldsymbol{K}} k$ iff $l \leq_{\boldsymbol{L}} \psi(k)$ for all elements $l \in L$ and $k \in K$. The closure property of bonds is obvious, since $l \mathit{bnd}(\boldsymbol{f}) = \uparrow_{\boldsymbol{K}} \varphi(l)$ for all elements $l \in L$ and $\mathit{bnd}(\boldsymbol{f}) k = \downarrow_{\boldsymbol{L}} \psi(k)$ for all elements $k \in K$.

  This can equivalently be defined in terms of either the left or the right monotonic function. Here we use the left monotonic function. In particular, we define the classification of the bond via the right operator that maps the left function to a classification relation in the presence of the underlying partial order of the target complete lattice.

```
(8) (CNG$function bond)
    (CNG$signature bond adjoint-pair CLS.BND$bond)
    (forall (?a (adjoint-pair ?a))
        (and (= (CLS.BND$source (bond ?a)) (LAT$classification (target ?a)))
             (= (CLS.BND$target (bond ?a)) (LAT$classification (source ?a)))
             (= (CLS.BND$classification (bond ?a))
                ((SET.FTN$right (LAT$underlying (target ?a))) (left ?a)))))
```

- This bond is the bond of the underlying adjoint pair. This fact could be used as a definition.

```
    (forall (?a (adjoint-pair ?a))
        (= (bond ?a)
           (ORD.ADJ$bond (underlying ?a))))
```

- The functor composition $B \circ A$ is naturally isomorphic to the identity functor $Id_{\text{Complete Adjoint}}$. To see this, let $\boldsymbol{L} = \langle L, \leq_{\boldsymbol{L}}, \wedge_{\boldsymbol{L}}, \vee_{\boldsymbol{L}} \rangle$ be a complete lattice with associated classification $B(\boldsymbol{L}) = \langle L, L, \leq_L \rangle$. Part of the fundamental theorem of concept lattices asserts the isomorphism $\boldsymbol{L} \cong A(B(\boldsymbol{L}))$. In particular, formal concepts of $A(B(\boldsymbol{L}))$ are cuts of the form $(\downarrow_{\boldsymbol{L}} x, \uparrow_{\boldsymbol{L}} x)$ for elements $x \in L$. The cut monotonic function $\boldsymbol{L} \to A(B(\boldsymbol{L}))$ was defined in the complete lattice namespace by $x \mapsto (\downarrow_{\boldsymbol{L}} x, \uparrow_{\boldsymbol{L}} x)$ for every element $x \in L$. This function is a bijection. Let $\langle \varphi, \psi \rangle : L \rightleftarrows K$ be a complete adjoint, an adjoint pair of monotonic functions, between complete lattices $L = \langle L, \leq_L, \wedge_L, \vee_L \rangle$, and $K = \langle K, \leq_K, \wedge_K, \vee_K \rangle$ with associated bond $B(\langle \varphi, \psi \rangle) : B(L) \to B(K)$. Then, the right adjoint of $A(B(\langle \varphi, \psi \rangle))$ maps $(\downarrow_L x, \uparrow_L x) \mapsto (\downarrow_K \psi(x), \uparrow_K \psi(x))$ and the left adjoint of $A(B(\langle \varphi, \psi \rangle))$ maps $(\downarrow_K y, \uparrow_K y) \mapsto (\downarrow_L \varphi(y), \uparrow_L \varphi(y))$. So, up to isomorphism, $A(B(\langle \varphi, \psi \rangle))$ is the same as $\langle \varphi, \psi \rangle$. This defines the natural isomorphism: $B \circ A \equiv Id_{\text{Complete Adjoint}}$.

  The *cut-forward* adjoint pair has the cut function as its right monotonic function and the composition of extent and join (or the composition of intent and meet) as its left monotonic function. This adjoint pair is a pair of inverse functions, and hence is an isomorphism from the complete lattice of its associated classification $A(B(\boldsymbol{L}))$ to a complete lattice $\boldsymbol{L}$. The *cut-reverse* adjoint pair flips the inverses – it has the cut function as its left monotonic function and the composition of extent and join as its right monotonic function. This adjoint pair is a pair of inverse functions, and hence is an isomorphism from a complete lattice $\boldsymbol{L}$ to the complete lattice of its associated classification $A(B(\boldsymbol{L}))$.

```
(9) (CNG$function cut-forward)
```

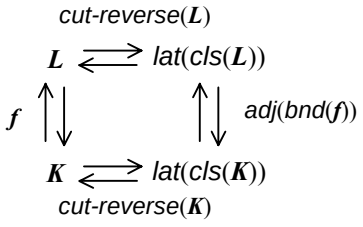
```
(CNG$signature cut-forward LAT$complete-lattice adjoint-pair)
(forall (?l (complete-lattice ?l))
    (and (= (source (cut-forward ?l))
            (CLS.CL$complete-lattice (LAT$classification ?l)))
         (= (target (cut-forward ?l)) ?l)
         (= (ORD.FTN$function (left (cut-forward ?l)))
            (SET.FTN$composition
                (CLS.CL$extent (LAT$classification ?l))
                (LAT$join ?l)))
         (= (ORD.FTN$function (right (cut-forward ?l))) (LAT$cut ?l))))
```

```
(10) (CNG$function cut-reverse)
     (CNG$signature cut-reverse LAT$complete-lattice adjoint-pair)
     (forall (?l (complete-lattice ?l))
         (and (= (source (cut-reverse ?l)) ?l)
              (= (target (cut-reverse ?l))
                 (CLS.CL$complete-lattice (LAT$classification ?l)))
              (= (ORD.FTN$function (left (cut-reverse ?l))) (LAT$cut ?l))
              (= (ORD.FTN$function (right (cut-reverse ?l)))
                 (SET.FTN$composition
                     (CLS.CL$extent (LAT$classification ?l))
                     (LAT$join ?l)))))
```

- The *composition* of two composable adjoint pairs $\boldsymbol{F} : \boldsymbol{A} \to \boldsymbol{B}$ and $\boldsymbol{G} : \boldsymbol{B} \to \boldsymbol{C}$ is the composition of the underlying adjoint pairs.

```
(11) (CNG$function composition)
     (CNG$signature composition adjoint-pair adjoint-pair adjoint-pair)
     (forall (?f (adjoint-pair ?f) ?g (adjoint-pair ?g))
         (<=> (exists (?h (adjoint-pair ?h)) (= (composition ?f ?g) ?h))
              (= (target ?f) (source ?g))))
     (forall (?f (adjoint-pair ?f) ?g (adjoint-pair ?g))
         (=> (= (target ?f) (source ?g))
             (and (= (source (composition ?f ?g)) (source ?f))
                  (= (target (composition ?f ?g)) (target ?g))
                  (= (underlying (composition ?f ?g))
                     (ORD.ADJ$composition (underlying ?f) (underlying ?g))))))
```

  - Let $\boldsymbol{f} = \langle \varphi, \psi \rangle : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$ be a complete adjoint, an adjoint pair of monotonic functions, between complete lattices $\boldsymbol{L} = \langle L, \leq_L, \wedge_L, \vee_L \rangle$, and $\boldsymbol{K} = \langle K, \leq_K, \wedge_K, \vee_K \rangle$ with associated bond $bnd(\boldsymbol{f}) : cls(\boldsymbol{L}) \to cls(\boldsymbol{K})$. Then, the right adjoint of $adj(bnd(\boldsymbol{f}))$ maps $(\downarrow_L x, \uparrow_L x) \mapsto (\downarrow_K \psi(x), \uparrow_K \psi(x))$ and the left adjoint of $adj(bnd(\boldsymbol{f}))$ maps $(\downarrow_K y, \uparrow_K y) \mapsto (\downarrow_L \varphi(y), \uparrow_L \varphi(y))$. This is equivalent to the natural isomorphism (commuting Diagram 3):



$$cut\text{-}reverse(\boldsymbol{L}) \cdot adj(bnd(\boldsymbol{f})) = \boldsymbol{f} \cdot cut\text{-}reverse(\boldsymbol{K}).$$

Diagram 3: Natural isomorphism

So, up to isomorphism, $adj(bnd(\boldsymbol{f}))$ is the same as $\boldsymbol{f}$. This defines the natural isomorphism: $B \circ A \cong Id_{\text{Complete Adjoint}}$.

```
(forall (?a (adjoint-pair ?a))
    (= (composition (cut-reverse (source ?a)) (adjoint-pair (bond ?a)))
       (composition ?a (cut-reverse (target ?a)))))
```

- The identity adjoint pair at a complete lattice $\boldsymbol{L}$ is the identity adjoint pair of the underlying order.

```
(12) (CNG$function identity)
     (CNG$signature identity LAT$complete-lattice adjoint-pair)
     (forall (?l (LAT$complete-lattice ?l))
         (and (= (source (identity ?l)) ?l)
              (= (target (identity ?l)) ?l)
              (= (underlying (identity ?l)) (ORD.ADJ$identity (LAT$underlying ?l)))))
```

- Duality can be extended to adjoint pairs. For any adjoint pair $\boldsymbol{f} : \boldsymbol{L} \rightleftarrows \boldsymbol{K}$, the *opposite* or *dual* of $\boldsymbol{f}$ is the adjoint pair $\boldsymbol{f}^{\perp} : \boldsymbol{K}^{\perp} \rightleftarrows \boldsymbol{L}^{\perp}$, whose source complete lattice is the opposite of the target of $\boldsymbol{f}$, whose target complete lattice is the opposite of the source of $\boldsymbol{f}$, whose underlying adjoint pair is the opposite of the adjoint pair of $\boldsymbol{f}$.

Axiom (11) specifies the opposite operator on adjoint pairs.

```
(13) (CNG$function opposite)
     (CNG$signature opposite adjoint-pair adjoint-pair)
     (forall (?f (adjoint-pair ?f))
         (and (= (source (opposite ?f)) (CL$opposite (target ?f)))
              (= (target (opposite ?f)) (CL$opposite (source ?f)))
              (= (underlying (opposite ?f)) (ORD.ADJ$opposite (underlying ?f)))))
```

- For any class function $f : A \to B$ there is a *power adjoint pair* $\wp f : \wp A \rightleftarrows \wp B$ over f defined as follows: the source is the complete lattice $\wp(A) = \langle \wp A, \subseteq_A, \cap_A, \cup_A \rangle$, the target is the complete lattice $\wp(B) = \langle \wp B, \subseteq_B, \cap_B, \cup_B \rangle$, the left monotonic function is the direct image function $\wp f : \wp A \to \wp B$, whose right monotonic function is the inverse image function $f^{-1} : \wp B \to \wp A$, and whose fundamental property holds, since the following holds.

  $\wp f(X) \subseteq_B Y$ iff $X \subseteq_A f^{-1}(Y)$

  for all subclasses $X \subseteq A$ and $Y \subseteq B$.

```
(14) (CNG$function power)
     (CNG$signature power SET.FTN$function adjoint-pair)
      (forall (?f (SET.FTN$function ?f))
          (and (= (source (power ?f)) (LAT$power (SET.FTN$source ?f)))
               (= (target (power ?f)) (LAT$power (SET.FTN$target ?f)))
               (= (left (power ?f)) (SET.FTN$direct-image ?f))
               (= (right (power ?f)) (SET.FTN$inverse-image ?f))))
```

  - Any adjoint pair $\boldsymbol{f} : \boldsymbol{L} = \langle L, \leq_L, \sqcap_L, \sqcup_L \rangle \rightleftarrows \boldsymbol{K} = \langle K, \leq_K, \sqcap_K, \sqcup_K \rangle$ between complete lattices is a concept morphism $\boldsymbol{f} : \langle \boldsymbol{K}, K, K, id_{\boldsymbol{K}}, id_{\boldsymbol{K}} \rangle \rightleftarrows \langle \boldsymbol{L}, L, L, id_{\boldsymbol{L}}, id_{\boldsymbol{L}} \rangle$ (in the reverse direction) between the concept lattices associated with the target and source complete lattices. Axiom (12) represents this as a CNG function. Note the contravariance.

```
(15) (CNG$function concept-morphism)
     (CNG$signature concept-morphism adjoint-pair CL.MOR$concept-morphism)
     (forall (?f (adjoint-pair ?f))
         (and (= (CL.MOR$source (concept-morphism ?f))
                 (LAT$concept-lattice (target ?f)))
              (= (CL.MOR$target (concept-morphism ?f))
                 (LAT$concept-lattice (source ?f)))
              (= (CL.MOR$adjoint-pair (concept-morphism ?f)) ?f)
              (= (CL.MOR$instance (concept-morphism ?f))
                 (ORD.ADJ$left (underlying ?f)))
              (= (CL.MOR$type (concept-morphism ?f))
                 (ORD.ADJ$right (underlying ?f)))))
```

- An easy check shows that the infomorphism of the concept morphism of an adjoint pair $\boldsymbol{f}$ is the same as the infomorphism associated with $\boldsymbol{f}$.

```
(forall (?f (adjoint-pair ?f))
    (= (CL.MOR$infomorphism (concept-morphism ?f))
       (infomorphism ?f)))
```

Complete Lattice Homomorphism

LAT.MOR

Unfortunately, adjoint pairs are not the best morphisms for making structural comparisons between complete lattices. Another morphism between complete lattices called complete homomorphisms are best for this.

- A *complete* (*lattice*) *homomorphism* $\psi : \boldsymbol{L} \to \boldsymbol{K}$ between complete lattices L and K is a (monotonic) function that preserves both joins and meets. Being meet-preserving, ψ has a left adjoint $\varphi : K \to L$, and being join-preserving ψ has a right adjoint $\theta : K \to L$. Therefore, a complete homomorphism is the middle monotonic function in two adjunctions $\varphi \dashv \psi \dashv \theta$. Since it is more algebraic, we use the latter

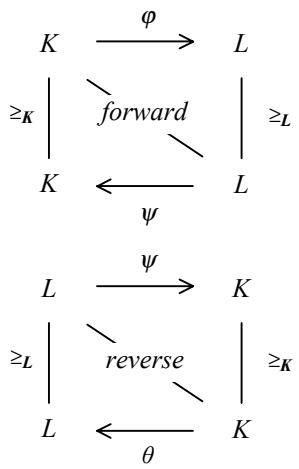


Figure 5: Complete Homomorphism

adjoint pair characterization in the definition of a complete lattice homomorphism.

```
(1) (CNG$conglomerate homomorphism)

(2) (CNG$function source)
    (CNG$signature source homomorphism LAT$complete-lattice)

(3) (CNG$function target)
    (CNG$signature target homomorphism LAT$complete-lattice)

(4) (CNG$function function)
    (CNG$signature function homomorphism ORD.FTN$monotonic-function)

(5) (CNG$function forward)
    (CNG$signature forward homomorphism LAT.ADJ$adjoint-pair)
    (forall (?h (homomorphism ?h))
        (and (= (LAT.ADJ$source (forward ?h)) (target ?h))
             (= (LAT.ADJ$target (forward ?h)) (source ?h))
             (= (LAT.ADJ$right (forward ?h)) (function ?h))))

(6) (CNG$function reverse)
    (CNG$signature reverse homomorphism LAT.ADJ$adjoint-pair)
    (forall (?h (homomorphism ?h))
        (and (= (LAT.ADJ$source (reverse ?h)) (source ?h))
             (= (LAT.ADJ$target (reverse ?h)) (target ?h))
             (= (LAT.ADJ$left (reverse ?h)) (function ?h))))
```

- The bond equivalent to a complete homomorphism would seem to be given by two bonds $\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{B}$ and $\boldsymbol{G} : \boldsymbol{B} \multimap \boldsymbol{A}$, where the right adjoint $\psi_{\boldsymbol{F}} : \boldsymbol{L}(\boldsymbol{A}) \to \boldsymbol{L}(\boldsymbol{B})$ of the complete adjoint $\boldsymbol{A}(\boldsymbol{F}) = \langle \varphi_{\boldsymbol{F}}, \psi_{\boldsymbol{F}} \rangle : \boldsymbol{L}(\boldsymbol{B}) \rightleftarrows \boldsymbol{L}(\boldsymbol{A})$ of the bond $\boldsymbol{F}$ is equal to the left adjoint $\varphi_{\boldsymbol{G}} : \boldsymbol{L}(\boldsymbol{A}) \to \boldsymbol{L}(\boldsymbol{B})$ of the complete adjoint $\boldsymbol{A}(\boldsymbol{G}) = \langle \varphi_{\boldsymbol{G}}, \psi_{\boldsymbol{G}} \rangle : \boldsymbol{L}(\boldsymbol{B}) \rightleftarrows \boldsymbol{L}(\boldsymbol{A})$ of the other bond $\boldsymbol{G}$ with the resultant adjunctions, $\varphi_{\boldsymbol{F}} \dashv \psi_{\boldsymbol{F}} = \varphi_{\boldsymbol{G}} \dashv \psi_{\boldsymbol{G}}$, where the middle adjoint is the complete homomorphism. This is indeed the case, but the question is what constraint to place on $\boldsymbol{F}$ and $\boldsymbol{G}$ in order for this to hold. The simple answer is to identify the actions of the two monotonic functions $\psi_{\boldsymbol{G}}$ and $\varphi_{\boldsymbol{F}}$. Let $(A, \Gamma) \in \boldsymbol{L}(\boldsymbol{A})$ be any formal concept in $\boldsymbol{L}(\boldsymbol{A})$. The action of the left adjoint $\varphi_{\boldsymbol{G}}$ on this concept is $(A, \Gamma) \mapsto (A^{\boldsymbol{GB}}, A^{\boldsymbol{G}})$, whereas the action of the right adjoint $\psi_{\boldsymbol{F}}$ on this concept is $(A, \Gamma) \mapsto (\Gamma^{\boldsymbol{F}}, \Gamma^{\boldsymbol{FB}})$. So the appropriate pointwise constraints are: $A^{\boldsymbol{GB}} = \Gamma^{\boldsymbol{F}}$ and $\Gamma^{\boldsymbol{FB}} = A^{\boldsymbol{G}}$, for every concept $(A, \Gamma) \in \boldsymbol{L}(\boldsymbol{A})$. The relational representation of these *pointwise constraints* is used in the definition of a bonding pair.

```
(7) (CNG$function bonding-pair)
    (CNG$signature bonding-pair homomorphism CLS.BNDPR$bonding-pair)
    (forall (?h (homomorphism ?h))
        (and (= (CLS.BND$source (bonding-pair ?h)) (LAT$classification (source ?a)))
             (= (CLS.BND$target (bonding-pair ?h)) (LAT$classification (target ?a)))
             (= (CLS.BND$forward (bonding-pair ?h))
                (LAT.ADJ$bond (forward ?h)))
             (= (CLS.BND$reverse (bonding-pair ?h))
                (LAT.ADJ$bond (reverse ?h)))))
```

- The functor composition $\boldsymbol{B}^2 \circ \boldsymbol{A}^2$ is naturally isomorphic to the identity functor $Id_{\text{Complete Lattice}}$. Consider any complete lattice $\boldsymbol{L}$. Any complete lattice $\boldsymbol{L}$ is isomorphic to the complete lattice $\boldsymbol{B}^2 \circ \boldsymbol{A}^2(\boldsymbol{L}) = \boldsymbol{L}(\langle \boldsymbol{L}, \boldsymbol{L}, \leq_{\boldsymbol{L}} \rangle)$ via the cut monotonic function $x \leftrightarrow (\downarrow_{\boldsymbol{L}} x, \uparrow_{\boldsymbol{L}} x)$. This function is bijective, and it preserves meets and joins. The *cut* complete lattice homomorphism $\psi : \boldsymbol{L} \to A^2(B^2(\boldsymbol{L})) = B^2 \circ A^2(\boldsymbol{L})$ is a bijection from a complete lattice $\boldsymbol{L}$ to the complete lattice of its classification $A^2(B^2(\boldsymbol{L}))$. Its forward adjoint pair is the cut-forward adjoint pair and its reverse adjoint pair is the cut-reverse adjoint pair.

```
(8) (CNG$function cut)
    (CNG$signature cut LAT$complete-lattice homomorphism)
    (forall (?l (complete-lattice ?l))
        (and (= (source (cut-forward ?l)) ?l)
             (= (target (cut-forward ?l))
                (CLS.CL$complete-lattice (LAT$classification ?l)))
             (= (forward (cut ?l))
                (LAT.ADJ$cut-forward ?l))
             (= (reverse (cut ?l)) (LAT.ADJ$cut-reverse ?l)))))
```

- Now consider any complete lattice homomorphism $\psi : \boldsymbol{L} \to \boldsymbol{K}$ between complete lattices $\boldsymbol{L}$ and $\boldsymbol{K}$ with associated adjunctions $\varphi \dashv \psi \dashv \theta$. The bonding pair functor maps this to the bonding pair $\boldsymbol{B}^2(\psi) = (\boldsymbol{B}(\langle \varphi, \psi \rangle), \boldsymbol{B}(\langle \psi, \theta \rangle))$, and the complete lattice functor maps this to the complete homomorphism $\boldsymbol{A}^2(\boldsymbol{B}^2(\psi)) = \tilde{\psi} : \boldsymbol{L}(\langle \boldsymbol{L}, \boldsymbol{L}, \leq_{\boldsymbol{L}} \rangle) \to \boldsymbol{L}(\langle \boldsymbol{K}, \boldsymbol{K}, \leq_{\boldsymbol{K}} \rangle)$ with associated adjunctions $\tilde{\varphi} \dashv \tilde{\psi} \dashv \tilde{\theta}$, where $\tilde{\varphi}((\downarrow_{\boldsymbol{K}} y, \uparrow_{\boldsymbol{K}} y)) = (\downarrow_{\boldsymbol{L}} \varphi(y), \uparrow_{\boldsymbol{L}} \varphi(y))$, $\tilde{\psi}((\downarrow_{\boldsymbol{L}} x, \uparrow_{\boldsymbol{L}} x)) = (\downarrow_{\boldsymbol{K}} \psi(x), \uparrow_{\boldsymbol{K}} \psi(x))$ and $\tilde{\theta}((\downarrow_{\boldsymbol{K}} y, \uparrow_{\boldsymbol{K}} y)) = (\downarrow_{\boldsymbol{L}} \theta(y), \uparrow_{\boldsymbol{L}} \theta(y))$. Clearly, the naturality condition holds between ψ and $\boldsymbol{B}^2 \circ \boldsymbol{A}^2(\psi)$.

```
(forall (?h (homomorphism ?h))
    (= (composition
           (cut (source ?h))
           (CLS.BNDPR$homomorphism (bonding-pair ?h)))
       (composition ?h (cut (target ?h)))))
```

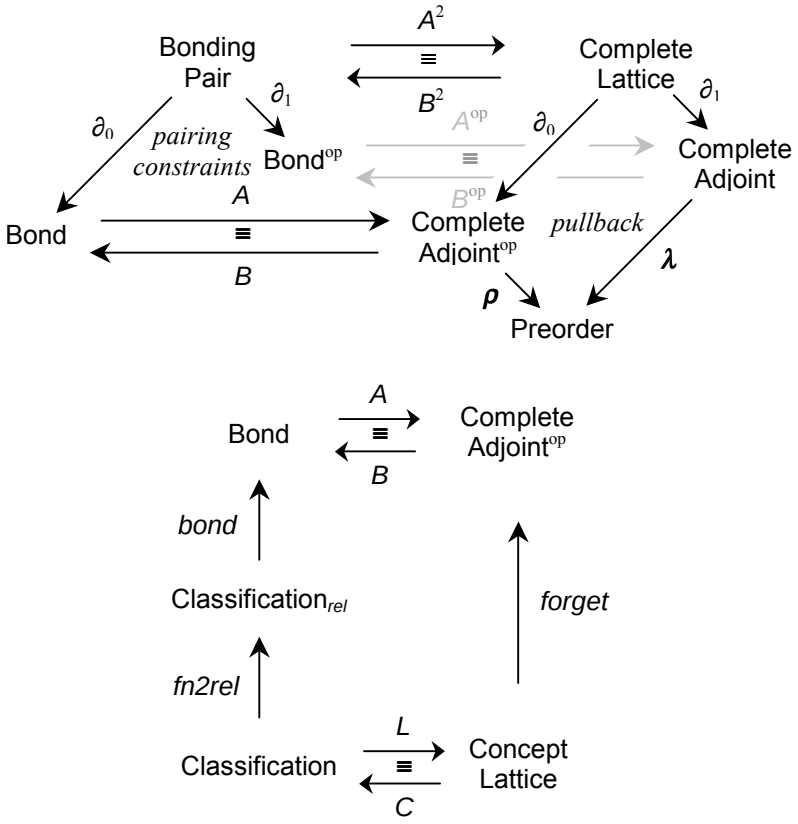


Figure 1: Architectural Diagram of Distributed Conceptual Structures (functors and natural isomorphisms)

Table 1: Mathematical–Ontological Correspondences in the Architectural Diagram

Mathematical Notation		Ontological Terminology
A^2	object part:	'CLS.CL$complete-lattice'
Complete Lattice Functor	morphism part:	'CLS.BNDPR$homomorphism'
B^2	object part:	'LAT$classification'
Bonding Pair Functor	morphism part:	'LAT.MOR$bonding-pair'
$Id_{\text{Complete Lattice}} \equiv B^2 \circ A^2$	component:	'LAT.MOR$cut'
Cut Natural Isomorphism		
$A^2 \circ B^2 \equiv Id_{\text{Bonding Pair}}$	component:	'CLS.BNDPR$iota-tau'
Iota-Tau Natural Isomorphism		
A	object part:	'CLS.CL$complete-lattice'
Complete Adjoint Functor	morphism part:	'CLS.BND$bond'
B	object part:	'LAT$classification'
Bond Functor	morphism part:	'LAT.ADJ$bond'
$Id_{\text{Complete Adjoint}} \equiv B \circ A$	component:	'LAT.ADJ$cut-reverse'
Cut-Reverse Natural Isomorphism		
$A \circ B \equiv Id_{\text{Bond}}$	component:	'CLS.BND$iota' (and 'CLS.BND$tau')
Iota Natural Isomorphism		
L	object part:	'CLS.CL$concept-lattice'
Concept Lattice Functor	morphism part:	'CLS.INFO$concept-morphism'
C	object part:	'CL$classification'
Classification Functor	morphism part:	'CL.MOR$infomorphism'
$C \circ L \equiv Id_{\text{Concept Lattice}}$	component:	'CL.MOR$representation'
Representation Natural Isomorphism		
$L \circ C = Id_{\text{Classification}}$		
equality		
∂_0	object part:	implicit identity function
0th Bond Projection Functor	morphism part:	'CLS.BNDPR$forward'
∂_1	object part:	implicit identity function
1st Bond Projection Functor	morphism part:	'CLS.BNDPR$reverse'
∂_0	object part:	implicit identity function
0th Adjoint Pair Projection Functor	morphism part:	'LAT.MOR$forward'
∂_1	object part:	implicit identity function
1st Adjoint Pair Projection Functor	morphism part:	'LAT.MOR$reverse'
$\boldsymbol{\lambda}$	object part:	'LAT$underlying'
Left Projection Functor	morphism part:	'LAT.ADJ$left'
$\boldsymbol{\rho}$	object part:	'LAT$underlying'
Right Projection Functor	morphism part:	'LAT.ADJ$right'

- The *complete lattice functor* A^2 : Bonding Pair → Complete Lattice is the operator that maps a classification $\boldsymbol{A}$ to its concept lattice $A^2(\boldsymbol{A}) \triangleq L(\boldsymbol{A})$ regarded as a complete lattice only, and maps a bonding pair $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \text{ -◆ } \boldsymbol{B}$ to its complete lattice homomorphism $A^2(\langle \boldsymbol{F}, \boldsymbol{G} \rangle) \triangleq \psi_F = \varphi_G : L(\boldsymbol{A}) \to L(\boldsymbol{B})$.
- The *bonding pair functor* B^2 : Complete Lattice → Bonding Pair is the operator that maps a complete lattice $\boldsymbol{L}$ to its classification $\langle L, L, \leq_L \rangle$ and maps a complete lattice homomorphism to its bonding pair as above. Since the bond functor $\boldsymbol{B}$ is functorial, so is B^2.

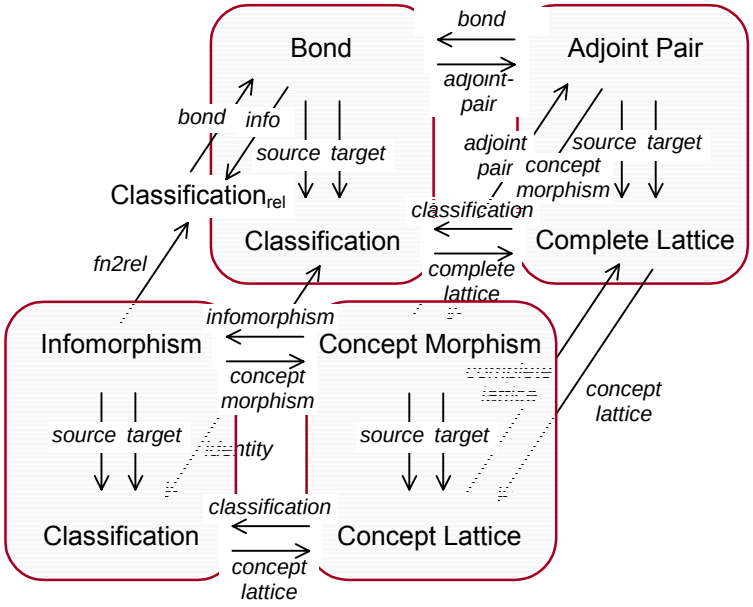


Diagram 2: Core Conglomerates and Functions in the Architectural Diagram

The Namespace of Large Classifications

This is the namespace for large classification and their morphisms: functional/relational infomorphisms, bonds and bonding pairs. In addition to strict classification terminology and axioms, this namespace will provide a bridge from classifications and their morphisms to complete/concept lattices and their morphisms. The terms introduced in this namespace are listed in Table 1.

Table 1: Terms introduced into the large classification namespace

	Conglomerate	Function	Example, Relation
CLS	'classification' 'separated' 'extensional'	'instance', 'type', 'incidence' 'extent', 'intent' 'indistinguishable', 'coextensive', 'opposite', 'instance-power'	
CLS .CL		'left-derivation', 'right-derivation' 'instance-closure', 'type-closure' 'concept', 'extent', 'intent', 'instance-generation', 'type-generation' 'concept-order', 'meet', 'join', '**complete-lattice**' 'coreflection', 'reflection' 'instance-embedding', 'type-embedding', 'instance-concept', 'type-concept' 'instance-order', 'type-order', '**concept-lattice**'	'truth-classification' 'truth-lattice'
CLS .FIB		'instance', 'instance-index' 'type', 'type-index' 'left-derivation', 'right-derivation' 'instance-closure', 'type-closure' 'concept', 'extent', 'intent', 'index' 'instance-generation', 'type-generation'	
CLS .COLL	'concept'	'classification', 'index' 'extent', 'intent' 'opposite' '2-cell', 'source', 'target' 'function', 'terminal' 'mediator-function' 'instance-distribution' 'type-distribution'	
CLS .INFO	'infomorphism'	'source', 'target', 'instance', 'type' '**bond**' 'monotonic-instance', 'monotonic-type' 'relational-instance', 'relational-type' '**relational-infomorphism**', '**fn2rel**' 'opposite', 'composition', 'identity' 'instance-power', 'eta' 'adjoint-pair', '**concept-morphism**'	
CLS .REL	'infomorphism'	'source', 'target', 'instance', 'type' '**bond**' 'opposite', 'composition', 'identity'	

CLS .BND	'bond'	'source', 'target', 'classification' '**bimodule**', '**infomorphism**', '**adjoint-pair**' 'opposite', 'composition', 'identity' '**iota**', '**tau**'	
CLS .BNDPR	'bonding-pair'	'source', 'target', 'forward', 'reverse' 'conceptual-image' 'opposite', 'composition', 'identity' '**tau-iota**', '**iota-tau**' '**homomorphism**'	
CLS .COL		'counique'	'initial'
CLS .COL .COPRD	'diagram' 'pair' 'cocone'	'classification1', 'classification2' 'instance-diagram', 'instance-pair' 'type-diagram', 'type-pair' 'opposite' 'cocone-diagram', 'opvertex', 'opfirst', 'opsecond' 'instance-cone', 'type-cocone' 'colimiting-cocone', 'colimit', 'binary- coproduct', 'injection1', 'injection2' 'comediator'	
CLS .COL .COINV	'coinvariant'	'classification', 'base', 'class', 'endorelation' 'coquotient', 'canon', 'comediator'	'respects'
CLS .COL .COEQ	'diagram' 'parallel-pair' 'cocone'	'source', 'target' 'infomorphism1', 'infomorphism2' 'instance-diagram', 'instance-parallel-pair' 'type-diagram', 'type-parallel-pair' 'coinvariant' 'cocone-diagram', 'opvertex', 'infomorphism' 'instance-cone', 'type-cocone' 'colimiting-cocone', 'colimit', 'coequalizer', 'canon'	
CLS .COL .PSH	'diagram' 'span' 'cocone'	'classification1', 'classification2', 'vertex', 'first', 'second' 'pair' 'opposite' 'instance-diagram', 'instance-opspan' 'type-diagram', 'type-span' 'coequalizer-diagram', 'parallel-pair' 'cocone-diagram', 'opvertex', 'opfirst', 'opsecond' 'binary-coproduct-cocone' 'coequalizer-cocone' 'colimiting-cocone', 'colimit', 'pushout', 'injection1', 'injection2', 'comediator'	

Table 2 lists the correspondence between standard mathematical notation and the ontological terminology in the namespace for classifications and functional/relation infomorphisms and bonds.

Table 2: Correspondence between Mathematical Notation and Ontological Terminology

Mathematical Notation	Ontological Terminology	Natural Language Description
CLS		
$\boldsymbol{A} = \langle \mathrm{tok}(\boldsymbol{A}), \mathrm{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$	‘classification’	a *classification* – identified with a binary relation; it is determined by its three components
$\mathrm{tok}(\boldsymbol{A})$	‘instance’	the *instance* (token) class of a classification – identified with the source class of a binary relation
$\mathrm{typ}(\boldsymbol{A})$	‘type’	the *type* class of a classification – identified with the target class of a binary relation
$\models_{\boldsymbol{A}}$	‘incidence’	the *incidence* (or *classification*) class of a classification – identified with the extent class of a binary relation
$\wp$	‘instance-power’	the *instance power* operator on classes – this maps classes to classifications
$i_1 \sim_{\boldsymbol{A}} i_2$	‘indistinguishable’	the information flow *indistinguishable* relation on instances
$t_1 \sim_{\boldsymbol{A}} t_2$	‘coextensive’	the information flow *coextensive* relation on types
“separated”	‘separated’	the *separated* subcollection of classifications
“extensional”	‘extensional’	the *extensional* subcollection of classifications
$(\text{-})^{\infty}$ or $(\text{-})^{\perp}$ or $(\text{-})^{\mathrm{op}}$	‘opposite’	the *involution* (or *transpose* or *opposite* or *dual*) operator on classifications
CLS.CL		
$(\text{-})'$ or $(\text{-})^{\boldsymbol{A}}$	‘left-derivation’, ‘right-derivation’	*left/right derivation* operations for a classification
$(\text{-})''$ or $(\text{-})^{\boldsymbol{A}\boldsymbol{A}}$	‘instance-closure’ ‘type-closure’	*instance/type closure* operations for a classification
$\boldsymbol{c} = (X, Y)$	‘concept’	a *formal concept* for a classification
$\boldsymbol{c}_1 \leq_{\boldsymbol{A}} \boldsymbol{c}_2$	‘concept-order’	the concept order of a classification – this is the partial order underlying the concept lattice of a classification
$\underline{\mathcal{B}}\boldsymbol{A}$	‘concept-lattice’	the *concept lattice* of a classification – the German word for “formal concepts” is *die begriffe*
$\gamma_{\boldsymbol{A}} : \mathrm{tok}(\boldsymbol{B}) \to \underline{\mathcal{B}}\boldsymbol{A}$	‘instance-embedding’	the *instance embedding function* – maps an instance to the concept that it generates
$\mu_{\boldsymbol{A}} : \mathrm{typ}(\boldsymbol{B}) \to \underline{\mathcal{B}}\boldsymbol{A}$	‘type-embedding’	the *type embedding function* – maps a type to the concept that it generates
CLS.INFO		
$f = \langle f^{\wedge}, f^{\vee} \rangle : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$	‘infomorphism’	an *infomorphism* from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ – determined by its instance and type functions
$f^{\wedge} : \mathrm{typ}(\boldsymbol{A}) \to \mathrm{typ}(\boldsymbol{B})$	‘type’	the *type* function of an infomorphism
$f^{\vee} : \mathrm{tok}(\boldsymbol{B}) \to \mathrm{tok}(\boldsymbol{A})$	‘instance’	the *instance* (or token) function of an infomorphism
$gf : \boldsymbol{A} \rightleftarrows \boldsymbol{C}$	‘composition’	the *composition* of two composable infomorphisms $f : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ and $g : \boldsymbol{B} \rightleftarrows \boldsymbol{C}$
$1_{\boldsymbol{A}} : \boldsymbol{A} \rightleftarrows \boldsymbol{A}$	‘identity’	the *identity* infomorphism on classification $\boldsymbol{A}$
$(\text{-})^{\infty}$ or $(\text{-})^{\perp}$ or $(\text{-})^{\mathrm{op}}$	‘opposite’	the *involution* (or *transpose* or *opposite* or *dual*) operator on infomorphisms
$\wp$	‘instance-power’	the *instance power* operator on functions – this maps functions to infomorphisms

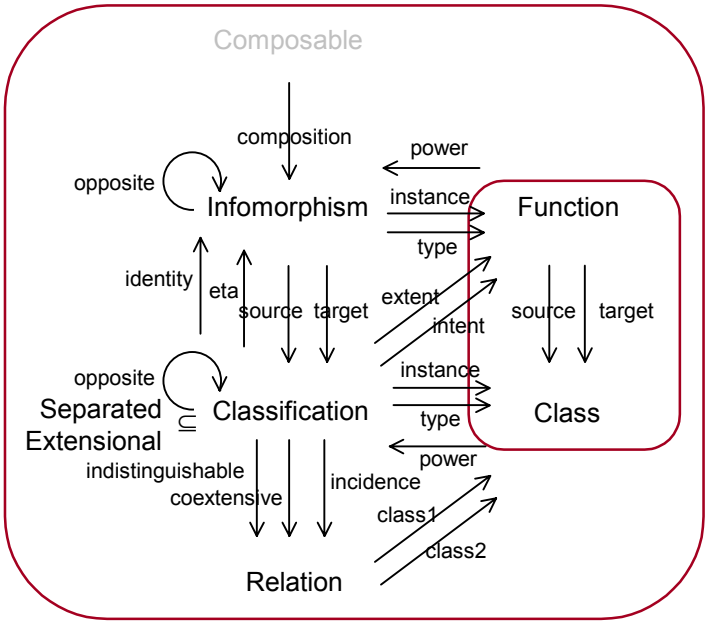


Diagram 3: Core Conglomerates and Functions for Classifications and Infomorphisms

Classifications

CLS

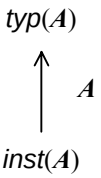


Figure 1: Classification

- A (large) *classification* $\boldsymbol{A} = \langle inst(\boldsymbol{A}), typ(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$ is identical to a (large) binary relation. It consists of a class of instances $inst(\boldsymbol{A})$ identified with the first or source class of a binary relation, a class of types $typ(\boldsymbol{A})$ identified with the second or target class of a binary relation, and a class of incidence or classification $\models_{\boldsymbol{A}}$ identified with the extent class of a binary relation. We visualize a binary relation as in Figure 1.

  The following is a KIF representation for the elements of a classification. Classifications are specified by declaration and population. The elements in the KIF representation are useful for the declaration and population of a classification. The term '`classification`' specified in axiom (1) allows one to *declare* classifications. The terms '`instance`' and '`type`' specified in axioms (2–3) and the term '`incidence`' specified in axiom (4) resolve classifications into their parts, thus allowing one to *populate* classifications.

```
(1) (CNG$conglomerate classification)
    (= classification REL$relation)

(2) (CNG$function instance)
    (CNG$signature instance classification CLS$class)
    (= instance REL$class1)

(3) (CNG$function type)
    (CNG$signature type classification CLS$class)
    (= type REL$class2)

(4) (CNG$function incidence)
    (CNG$signature incidence classification CLS$class)
    (= incidence REL$extent)
```

- Associated with any classification is a function that produces the *extent* of a type and a function that produces the *intent* of an instance, both within the context of the classification. The extent of a type $t \in typ(\boldsymbol{A})$ in a classification $\boldsymbol{A}$ defined by

$extent_A(t) = \{i \in inst(\boldsymbol{A}) \mid i \vDash_A t\}$.

- Dually, the intent of an instance $i \in inst(\boldsymbol{A})$ in a classification $\boldsymbol{A} = \langle inst(\boldsymbol{A}), typ(\boldsymbol{A}), \vDash_A \rangle$ is defined by $intent_A(i) = \{t \in typ(\boldsymbol{A}) \mid i \vDash_A t\}$.

Axiom (5) specifies the extent function and axiom (6) specifies the intent function. The last axiom in (5) demonstrates that the relative instantiation-predication represented by the incidence relation is compatible with, generalizes and relativizes the absolute KIF instantiation-predication – an instance is a member of the extent of a type iff the instance is classified by the type.

```
(5) (CNG$function extent)
    (CNG$signature extent classification SET.FTN$function)
    (forall (?a (classification ?a))
        (= (SET.FTN$source (extent ?a)) (type ?a))
        (= (SET.FTN$target (extent ?a)) (SET$power (instance ?a)))
        (forall (?i ?t ((instance ?a) ?i) ((type ?a) ?t))
            (<=> (((extent ?a) ?t) ?i)
                 ((incidence ?a) [?i ?t]))))

(6) (CNG$function intent)
     (CNG$signature intent classification SET.FTN$function)
     (forall (?a (classification ?a))
         (= (SET.FTN$source (intent ?a)) (instance ?a))
         (= (SET.FTN$target (intent ?a)) (SET$power (type ?a)))
         (forall (?i ?t ((instance ?a) ?i) ((type ?a) ?t))
             (<=> (((intent ?a) ?i) ?t)
                  ((REL$extent (incidence ?a)) [?i ?t]))))
```

- For any classification $\boldsymbol{A} = \langle inst(\boldsymbol{A}), typ(\boldsymbol{A}), \vDash_A \rangle$, two instances $i_1, i_2 \in inst(\boldsymbol{A})$ are *indistinguishable* in $\boldsymbol{A}$ (Barwise and Seligman, 1997), written $i_1 \sim_A i_2$, when $intent_A(i_1) = intent_A(i_2)$. Two types $t_1, t_2 \in typ(\boldsymbol{A})$ are *coextensive* in $\boldsymbol{A}$, written $t_1 \sim_A t_2$, when $extent_A(t_1) = extent_A(t_2)$. A classification A is *separated* when there are no distinct indistinguishable instances, and *extensional* when there are no distinct coextensive types.

The terms '`(indistinguishable ?a)`' and '`(coextensive ?a)`' that are specified in axioms (7–8) represent the Information Flow notions of instance *indistinguishability* and type *coextension*, respectively. The terms '`separated`' and '`extensional`' that are specified in axioms (9–10) represent the Information Flow notions of classification *separateness* and *extensionality*, respectively.

```
(7) (CNG$function indistinguishable)
    (CNG$signature indistinguishable classification REL.ENDO$relation)
    (forall (?a (classification ?a))
        (and (= (REL$class (indistinguishable ?a)) (instance ?a))
             (forall (?i1 ((instance ?a) ?i1)
                      ?i2 ((instance ?a) ?i2))
                 (<=> ((REL$extent (indistinguishable ?a)) [?i1 ?i2])
                      (= ((intent ?a) ?i1) ((intent ?a) ?i2)))))))

(8) (CNG$function coextensive)
    (CNG$signature coextensive classification REL.ENDO$relation)
    (forall (?a (classification ?a))
        (and (= (REL$class (coextensive ?a)) (type ?a))
             (forall (?t1 ((type ?a) ?t1)
                      ?t2 ((type ?a) ?t2))
                 (<=> ((REL$extent (coextensive ?a)) [?t1 ?t2])
                      (= ((extent ?a) ?t1) ((extent ?a) ?t2)))))))

(9) (CNG$conglomeration separated)
    (CNG$subconglomeration separated classification)
    (forall (?a (classification ?a))
        (<=> (separated ?a)
             (REL.ENDO$subendorelation
                 (indistinguishable ?a)
                 (REL.ENDO$identity (instance ?a))))))

(10) (CNG$conglomeration extensional)
     (CNG$subconglomeration extensional classification)
```

```
(forall (?a (classification ?a))
    (<=> (extensional ?a)
        (REL.ENDO$subendorelation
            (coextensive ?a)
            (REL.ENDO$identity (type ?a)))))
```

- To quote (Barwise and Seligman, 1997), "in any classification, we think of the types as classifying the instances, but it is often useful to think of the instances as classifying the types." For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, the *opposite* or *dual* of $\boldsymbol{A}$ is the opposite binary relation; that is, the classification $\boldsymbol{A}^{\perp} = \langle \mathit{typ}(\boldsymbol{A}), \mathit{inst}(\boldsymbol{A}), \models_{\boldsymbol{A}}^{\perp} \rangle$, whose instances are types of $\boldsymbol{A}$, and whose types are instances of $\boldsymbol{A}$, and whose incidence is: $t \models^{\perp} i$ when $i \models t$. Axiom (11) specifies the opposite operator on classifications.

```
(11) (CNG$function opposite)
     (CNG$signature opposite classification classification)
     (= opposite REL$opposite)
```

- For any class A the *instance power classification* $\wp A = \langle A, \wp A, \in_A \rangle$ over A are defined as follows: the instance class is A; the type class is the power class $\wp A$ (so that a type is a subclass of A), and the incidence relation is the membership relation $\in_A$. Axiom (12) specifies the power operator from classes to classifications.

```
(12) (CNG$function instance-power)
     (CNG$signature instance-power SET$class classification)
     (forall (?a (SET$class ?a))
         (and (= (instance (instance-power ?a)) ?a)
              (= (type (instance-power ?a)) (SET$power ?a))
              (forall (?x ?y (?a ?x) ((SET$power ?a) ?y))
                  (<=> ((incidence (instance-power ?a)) [?x ?y])
                       (?y ?x))))))
```

Concept Lattices

CLS.CL

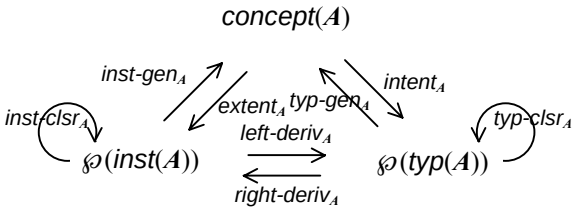


Diagram 4: Core classes/functions for concept lattices

○ For any large classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$ there are two senses of *derivation* corresponding to the two senses of relational residuation. Left derivation maps a subset of instances to the types on which they are all incident, and dually right derivation maps a subset of types to the instances, which are incident on all of them. For all $X \subseteq \mathit{inst}(\boldsymbol{A})$ and $Y \subseteq \mathit{typ}(\boldsymbol{A})$

$$X \mapsto X' = X \backslash \boldsymbol{A} = \{t \in \mathit{typ}(\boldsymbol{A}) \mid i \models t \text{ for all } i \in X\}$$

$$Y \mapsto Y' = \boldsymbol{A} / Y = \{i \in \mathit{inst}(\boldsymbol{A}) \mid i \models t \text{ for all } t \in Y\}.$$

```
(1) (CNG$function left-derivation)
    (CNG$signature left-derivation CLS$classification SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (left-derivation ?a))
                (SET$power (CLS$instance ?a)))
             (= (SET.FTN$target (left-derivation ?a))
                (SET$power (CLS$type ?a)))
             (forall (?x (SET$subclass ?x (CLS$instance ?a)))
                     ?t ((CLS$type ?a) ?t))
                 (<=> (((left-derivation ?a) ?x) ?t)
                      (forall (?i ((CLS$instance ?a) ?i))
                          (=> (?x ?i) ((CLS$incidence ?a) [?i ?t])))))))

(2) (CNG$function right-derivation)
    (CNG$signature right-derivation CLS$classification SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (right-derivation ?a))
                (SET$power (CLS$type ?a)))
             (= (SET.FTN$target (right-derivation ?a))
                (SET$power (CLS$instance ?a)))
             (forall (?y (SET$subclass ?y (CLS$type ?a)))
                     ?i ((CLS$instance ?a) ?i))
                 (<=> (((right-derivation ?a) ?y) ?i)
                      (forall (?t ((CLS$type ?a) ?t))
                          (=> (?y ?t) ((CLS$incidence ?a) [?i ?t])))))))
```

○ A simple fundamental result is the following equivalence. For all $X \subseteq \mathit{inst}(\boldsymbol{A})$ and $Y \subseteq \mathit{typ}(\boldsymbol{A})$

$$Y \subseteq X \backslash \boldsymbol{A} \text{ in } \wp(\mathit{typ}(\boldsymbol{A}))^{\mathrm{op}} \text{ iff } X \times Y \subseteq \models_{\boldsymbol{A}} \text{ iff } X \subseteq \boldsymbol{A} / Y \text{ in } \wp(\mathit{inst}(\boldsymbol{A})).$$

This describes a Galois connection (preorder adjunction) between the left derivation

$$(\text{-}) \backslash \boldsymbol{A} : \wp(\mathit{inst}(\boldsymbol{A})) \to \wp(\mathit{typ}(\boldsymbol{A}))^{\mathrm{op}}$$

and the right derivation

$$\boldsymbol{A} / (\text{-}) : \wp(\mathit{typ}(\boldsymbol{A}))^{\mathrm{op}} \to \wp(\mathit{inst}(\boldsymbol{A})).$$

The first two facts assert (contravariant) monotonicity of derivation. The last fact asserts the adjointness condition.

```
(forall (?x1 (SET$subclass ?x1 (CLS$instance ?a)
         ?x2 (SET$subclass ?x2 (CLS$instance ?a))
    (=> (SET$subclass ?x1 ?x2)
        (SET$subclass ((left-derivation ?a) ?x2) ((left-derivation ?a) ?x1))))

(forall (?y1 (SET$subclass ?y1 (CLS$type ?a)
         ?y2 (SET$subclass ?y2 (CLS$type ?a))
    (=> (SET$subclass ?y2 ?y1)
        (SET$subclass ((right-derivation ?a) ?y1) ((right-derivation ?a) ?y2))))
```

```
(forall (?x (SET$subclass ?x (CLS$instance ?a))
         ?y (SET$subclass ?y (CLS$type ?a)))
    (<=> (SET$subclass ?x ((right-derivation ?a) ?y))
         (SET$subclass ?y ((left-derivation ?a) ?x))))
```

- The composition of derivations gives two senses of closure operator.

```
(3) (CNG$function instance-closure)
    (CNG$signature instance-closure CLS$classification SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (instance-closure ?a))
                (SET$power (CLS$instance ?a)))
             (= (SET.FTN$target (instance-closure ?a))
                (SET$power (CLS$instance ?a)))
             (= (instance-closure ?a)
                (SET.FTN$composition (left-derivation ?a) (right-derivation ?a))))))

(4) (CNG$function type-closure)
    (CNG$signature type-closure CLS$classification SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (type-closure ?a))
                (SET$power (CLS$type ?a)))
             (= (SET.FTN$target (type-closure ?a))
                (SET$power (CLS$type ?a)))
             (= (type-closure ?a)
                (SET.FTN$composition (right-derivation ?a) (left-derivation ?a))))))
```

- The following (easily proven) results confirm that these are closure operators.

  $X \subseteq X''$ and $X' = X'''$ for all $X \subseteq \mathit{inst}(\boldsymbol{A})$.

  $Y \subseteq Y''$ and $Y' = Y'''$ for all $Y \subseteq \mathit{typ}(\boldsymbol{A})$.

```
(forall (?a (CLS$classification ?a))
    (and (= (left-derivation ?a)
            (SET.FTN$composition (instance-closure ?a) (left-derivation ?a)))
         (forall (?x (SET$subclass ?x (CLS$instance ?a))
             (SET$subclass ?x ((instance-closure ?a) ?x))))))

(forall (?a (CLS$classification ?a))
    (and (= (right-derivation ?a)
            (SET.FTN$composition (type-closure ?a) (right-derivation ?a)))
         (forall (?y (SET$subclass ?y (CLS$type ?a))
             (SET$subclass ?y ((type-closure ?a) ?y))))))
```

- Further properties of Galois connection relate to continuity – the closure of a union of a family of classes is the intersection of the closures.

  $(\cup_{j \in J} X_j)' = \cap_{j \in J} X_j'$ for any family of subsets $X_j \subseteq \mathit{inst}(\boldsymbol{A})$ for $j \in J$.

  $(\cup_{k \in K} Y_k)' = \cap_{k \in K} Y_k'$ for any family of subsets $Y_k \subseteq \mathit{typ}(\boldsymbol{A})$ for $k \in K$.

- By embedding the subsets of instances and types above as relations, we can connect derivation to residuation. We can prove the following simple identities: the embedding of the left-derivation of a subset of instances is the left-residuation of the classification along the opposite of the embedding of the subset, and dually for the right notions.

```
(forall (?a (CLS$classification ?a))
    (and (forall (?x (SET$subclass ?x (CLS$instance ?a)))
             (= ((REL$embed (CLS$type ?a)) ((left-derivation ?a) ?x))
                (REL$left-residuation
                    (REL$opposite ((REL$embed (CLS$instance ?a)) ?x))
                    ?a))
         (forall (?y (SET$subclass ?y (CLS$type ?a)))
             (= (REL$opposite
                    ((REL$embed (CLS$instance ?a)) ((right-derivation ?a) ?y)))
                (REL$right-residuation
                    ((REL$embed (CLS$type ?a)) ?y)
                    ?a))))))
```

- For a classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, the closed elements of the derivation Galois connection are called formal concepts. There are several ways to define this notion, but traditionally it has been given the following (slightly redundant) definition. A (large) *formal concept* $c = \langle \mathit{extent}_{\boldsymbol{A}}(c), \mathit{intent}_{\boldsymbol{A}}(c) \rangle$ is a pair of classes, $\mathit{extent}_{\boldsymbol{A}}(c) \subseteq \mathit{inst}(\boldsymbol{A})$ and $\mathit{intent}_{\boldsymbol{A}}(c) \subseteq \mathit{typ}(\boldsymbol{A})$, that satisfy the equivalent conditions that $\mathit{extent}_{\boldsymbol{A}}(c) = \mathit{intent}_{\boldsymbol{A}}(c)'$ and $\mathit{intent}_{\boldsymbol{A}}(c) = \mathit{extent}_{\boldsymbol{A}}(c)'$. Let $\mathit{concept}(\boldsymbol{A})$ denote the class of all formal concepts of a classification $\boldsymbol{A}$. There are functions, $\mathit{extent}_{\boldsymbol{A}} : \mathit{concept}(\boldsymbol{A}) \to \wp(\mathit{inst}(\boldsymbol{A}))$ and $\mathit{intent}_{\boldsymbol{A}} : \mathit{concept}(\boldsymbol{A}) \to \wp(\mathit{typ}(\boldsymbol{A}))$, that map concepts to their extent and intent.

```
(5) (CNG$function concept)
    (CNG$signature concept CLS$classification SET$class)

(6) (CNG$function extent)
    (CNG$signature extent CLS$classification SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (extent ?a)) (concept ?a))
             (= (SET.FTN$target (extent ?a)) (SET$power (CLS$instance ?a)))))

(7) (CNG$function intent)
    (CNG$signature intent CLS$classification SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (intent ?a)) (concept ?a))
             (= (SET.FTN$target (intent ?a)) (SET$power (CLS$type ?a)))))

    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$composition (intent ?a) (right-derivation ?a))
                (extent ?a))
             (= (SET.FTN$composition (extent ?a) (left-derivation ?a))
                (intent ?a))))
```

- There are surjective generator functions, called instance-generation and type-generation, that map subsets of instances and types to their generated concepts. For all $X \subseteq \mathit{inst}(\boldsymbol{A})$ and $Y \subseteq \mathit{typ}(\boldsymbol{A})$

  $X \mapsto \langle X'', X' \rangle$ "instance-generation"

  $Y \mapsto \langle Y', Y'' \rangle$ "type-generation".

  See diagram 4 for a visualization of the following conditions.
  - The composition of instance-generation and intent equals left-derivation, and the composition of type-generation and extent equals right-derivation.
  - The composition of instance-generation and extent equals instance-closure, and the composition of type-generation and intent equals type-closure.
  - The composition of extent and instance-generation is identity, and the composition of intent and type-generation is identity.

  These conditions define generation, and also insure that all concepts are captured in the class '`(concept ?a)`'. Concepts are determined by either their extents or their intents – this is represented by the fact that the extent and intent functions are injective.

```
(8) (CNG$function instance-generation)
    (CNG$signature instance-generation CLS$classification SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (instance-generation ?a))
                (SET$power (CLS$instance ?a)))
             (= (SET.FTN$target (instance-generation ?a))
                (concept ?a))
             (= (SET.FTN$composition (instance-generation ?a) (intent ?a))
                (left-derivation ?a))
             (= (SET.FTN$composition (instance-generation ?a) (extent ?a))
                (instance-closure ?a))
             (= (SET.FTN$composition (extent ?a) (instance-generation ?a))
                (SET.FTN$identity (concept ?a)))))

(9) (CNG$function type-generation)
    (CNG$signature type-generation CLS$classification SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (= (SET.FTN$source (type-generation ?a))
                (SET$power (CLS$type ?a)))
```

```
            (= (SET.FTN$target (type-generation ?a))
               (concept ?a))
            (= (SET.FTN$composition (type-generation ?a) (extent ?a))
               (right-derivation ?a))
            (= (SET.FTN$composition (type-generation ?a) (intent ?a))
               (type-closure ?a))
            (= (SET.FTN$composition (intent ?a) (type-generation ?a))
               (SET.FTN$identity (concept ?a)))))
```

- A concept $c_1 = \langle extent_A(c_1), intent_A(c_1) \rangle$ is a *subconcept* of a concept $c_2 = \langle extent_A(c_2), intent_A(c_2) \rangle$, denoted $c_1 \leq_A c_2$, when $extent_A(c_1) \subseteq extent_A(c_2)$ or equivalently when $intent_A(c_1) \supseteq intent_A(c_2)$. Other language is that c_1 is "more specific" than c_2, and c_2 is "more generic" than c_1. For any classification $\boldsymbol{A} = \langle inst(\boldsymbol{A}), typ(\boldsymbol{A}), \models_A \rangle$, the class of concepts together with the subconcept relation form a partial order $order(\boldsymbol{A}) = \langle concept(\boldsymbol{A}), \leq_A \rangle$.

```
(10) (CNG$function concept-order)
     (CNG$signature concept-order CLS$classification ORD$partial-order)
     (forall (?a (CLS$classification ?a))
         (and (= (ORD$class (concept-order ?a)) (concept ?a))
              (forall (?c1 ((concept ?a) ?c1)
                       ?c2 ((concept ?a) ?c2))
                  (<=> ((ORD$extent (concept-order ?a)) [?c1 ?c2])
                       (SET$subclass ((extent ?a) ?c1) ((extent ?a) ?c2)))))))
```

- There are both a meet and a join operations defined on subclasses of concepts

  $meet_A : \wp(concept(\boldsymbol{A})) \rightarrow concept(\boldsymbol{A})$

  $join_A : \wp(concept(\boldsymbol{A})) \rightarrow concept(\boldsymbol{A})$

  defined as follows:

  $$\sqcap_L(C) = \sqcap_{c \in C} \langle extent_A(c), intent_A(c) \rangle = \langle \cap_{c \in C} extent_A(c), (\cup_{c \in C} intent_A(c))'' \rangle$$

  $$\sqcup_L(C) = \sqcup_{c \in C} \langle extent_A(c), intent_A(c) \rangle = \langle (\cup_{c \in C} extent_A(c))'', \cap_{c \in C} intent_A(c) \rangle$$

  for any subclass $C \subseteq concept(\boldsymbol{A})$.

```
(11) (CNG$function meet)
     (CNG$signature meet CLS$classification SET.FTN$function)
     (forall (?a (CLS$classification ?a))
         (and (= (SET.FTN$source (meet ?a)) (SET$power (concept ?a)))
              (= (SET.FTN$target (meet ?a)) (concept ?a))
              (forall (?c (SET$subclass ?c (concept ?a)))
                  (and (= (SET.FTN$composition (meet ?a) (extent ?a))
                          (SET.FTN$composition
                              (SET.FTN$power (extent ?a))
                              (SET$intersection (instance ?a))))
                       (= (SET.FTN$composition (meet ?a) (intent ?a))
                          (SET.FTN$composition
                              (SET.FTN$composition
                                  (SET.FTN$power (intent ?a))
                                  (SET$union (type ?a)))
                              (type-closure ?a)))))))

(12) (CNG$function join)
     (CNG$signature join CLS$classification SET.FTN$function)
     (forall (?a (CLS$classification ?a))
         (and (= (SET.FTN$source (join ?a)) (SET$power (concept ?a)))
              (= (SET.FTN$target (join ?a)) (concept ?a))
              (forall (?c (SET$subclass ?c (concept ?a)))
                  (and (= (SET.FTN$composition (join ?a) (extent ?a))
                          (SET.FTN$composition
                              (SET.FTN$composition
                                  (SET.FTN$power (extent ?a))
                                  (SET$union (instance ?a)))
                              (instance-closure ?a)))
                       (= (SET.FTN$composition (join ?a) (intent ?a))
                          (SET.FTN$composition
```

```
                (SET.FTN$power (intent ?a))
                (SET$intersection (type ?a))))))))
```

- For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, the partial order $\mathit{order}(\boldsymbol{A}) = \langle \mathit{concept}(\boldsymbol{A}), \leq_{\boldsymbol{A}} \rangle$ of concepts together with the conceptual join and meet operators form a complete lattice $\mathit{latt}(\boldsymbol{A}) = \langle \mathit{order}(\boldsymbol{A}), \mathit{join}(\boldsymbol{A}), \mathit{meet}(\boldsymbol{A}) \rangle$.

```
(13) (CNG$function complete-lattice)
     (CNG$signature complete-lattice CLS$classification LAT$complete-lattice)
     (forall (?a (CLS$classification ?a))
         (and (= (LAT$underlying (complete-lattice ?a)) (concept-order ?a))
              (= (LAT$join (complete-lattice ?a)) (join ?a))
              (= (LAT$meet (complete-lattice ?a)) (meet ?a))))
```

- For any two classifications $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}, \models_{\boldsymbol{A}} \rangle$ and $\boldsymbol{C} = \langle \mathit{inst}(\boldsymbol{C}), \mathit{typ}, \models_{\boldsymbol{C}} \rangle$ that have the same class of types and where the classification (binary relation) $\boldsymbol{C}$ is *type-closed* $(\boldsymbol{A}/\boldsymbol{C})\backslash\boldsymbol{A} = \boldsymbol{C}$ with respect to $\boldsymbol{A}$, there is an associated *coreflection* (adjoint pair) $\mathit{int}_{\boldsymbol{C},\boldsymbol{A}} = \langle \partial_0, \tilde{\partial}_0 \rangle : \mathit{latt}(\boldsymbol{C}) \rightleftarrows \mathit{latt}(\boldsymbol{A})$ between their complete lattices, where $\tilde{\partial}_0 : \mathit{latt}(\boldsymbol{A}) \to \mathit{latt}(\boldsymbol{C})$ is right adjoint right inverse (rari) to $\partial_0 : \mathit{latt}(\boldsymbol{C}) \to \mathit{latt}(\boldsymbol{A})$. Hence, ∂_0 embeds $\mathit{latt}(\boldsymbol{C})$ as an internal part of $\mathit{latt}(\boldsymbol{A})$. These functions are defined as follows.

  $\partial_0(\langle X, Y \rangle) = (\langle Y', Y \rangle)$ for any concept $\langle X, Y \rangle \in \mathit{latt}(\boldsymbol{C})$
  $\tilde{\partial}_0(\langle X, Y \rangle) = (\langle Y', Y'' \rangle)$ for any concept $\langle X, Y \rangle \in \mathit{latt}(\boldsymbol{A})$

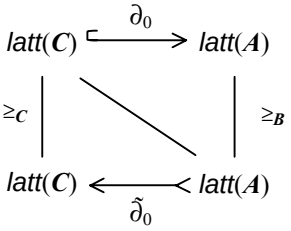


Figure 2: Type-closed adjoint pair

- Dually, for any two classifications $\boldsymbol{C} = \langle \mathit{inst}, \mathit{typ}(\boldsymbol{C}), \models_{\boldsymbol{C}} \rangle$ and $\boldsymbol{B} = \langle \mathit{inst}, \mathit{typ}(\boldsymbol{B}), \models_{\boldsymbol{B}} \rangle$ that have the same class of instances and where the classification (binary relation) $\boldsymbol{C}$ is *instance-closed* $\boldsymbol{B}/(\boldsymbol{C}\backslash\boldsymbol{B}) = \boldsymbol{C}$ with respect to $\boldsymbol{B}$, there is an associated *reflection* (adjoint pair) $\mathit{ext}_{\boldsymbol{B},\boldsymbol{C}} = \langle \tilde{\partial}_1, \partial_1 \rangle : \mathit{latt}(\boldsymbol{B}) \rightleftarrows \mathit{latt}(\boldsymbol{C})$ between their complete lattices, where $\tilde{\partial}_1 : \mathit{latt}(\boldsymbol{B}) \to \mathit{latt}(\boldsymbol{C})$ is left adjoint right inverse (lari) to $\partial_1 : \mathit{latt}(\boldsymbol{C}) \to \mathit{latt}(\boldsymbol{B})$. Hence, ∂_1 embeds $\mathit{latt}(\boldsymbol{C})$ as an external part of $\mathit{latt}(\boldsymbol{A})$. These functions are defined as follows.

  $\partial_1(\langle X, Y \rangle) = (\langle X, X' \rangle)$ for any concept $\langle X, Y \rangle \in \mathit{latt}(\boldsymbol{C})$
  $\tilde{\partial}_1(\langle X, Y \rangle) = (\langle X'', X' \rangle)$ for any concept $\langle X, Y \rangle \in \mathit{latt}(\boldsymbol{B})$.

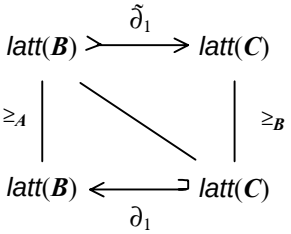


Figure 3: Instance-closed adjoint pair

```
(14) (CNG$function coreflection)
     (CNG$signature coreflection
         CLS$classification CLS$classification LAT.ADJ$adjoint-pair)
     (forall (?c (CLS$classification ?c)
              ?a (CLS$classification ?a))
         (<=> (exists (?f (LAT.ADJ$adjoint-pair ?f))
                  (= (coreflection ?c ?a) ?f))
              (and (= (CLS$type ?c) (CLS$type ?a))
                   (= (REL$left-residuation (REL$right-residuation ?c ?a) ?a) ?c))))
     (forall (?c (CLS$classification ?c)
              ?a (CLS$classification ?a))
         (=> (and (= (CLS$type ?c) (CLS$type ?a))
                  (= (REL$left-residuation (REL$right-residuation ?c ?a) ?a) ?c))
             (and (= (LAT.ADJ$source (coreflection ?c ?a))
                     (complete-lattice c?))
                  (= (LAT.ADJ$target (coreflection ?c ?a))
                     (complete-lattice ?a))
                  (= (REL.FTN$composition (LAT.ADJ$left (coreflection ?c ?a)) (extent ?a))
                     (REL.FTN$composition (intent ?c) (right-derivation ?c)))
                  (= (REL.FTN$composition (LAT.ADJ$left (coreflection ?c ?a)) (intent ?a))
                     (intent ?c))
                  (= (REL.FTN$composition (LAT.ADJ$right (coreflection ?c ?a)) (extent ?c))
                     (REL.FTN$composition (intent ?a) (right-derivation ?a)))
                  (= (REL.FTN$composition (LAT.ADJ$right (coreflection ?c ?a)) (intent ?c))
                     (REL.FTN$composition (intent ?a) (type-closure ?a))))))
```

```
(15) (CNG$function reflection)
     (CNG$signature reflection
         CLS$classification CLS$classification LAT.ADJ$adjoint-pair)
     (forall (?b (CLS$classification ?b)
              ?c (CLS$classification ?c))
         (<=> (exists (?f (LAT.ADJ$adjoint-pair ?f))
                  (= (reflection ?b ?c) ?f))
              (and (= (CLS$instance ?b) (CLS$instance ?c))
                   (= (REL$right-residuation (REL$left-residuation ?c ?b) ?b) ?c))))
     (forall (?b (CLS$classification ?b)
              ?c (CLS$classification ?c))
         (=> (and (CLS$instance ?b) (= (CLS$instance ?c))
                  (= (REL$right-residuation (REL$left-residuation ?c ?b) ?b) ?c))
             (and (= (LAT.ADJ$source (reflection ?b ?c))
                     (complete-lattice b?))
                  (= (LAT.ADJ$target (reflection ?b ?c))
                     (complete-lattice c?))
                  (= (REL.FTN$composition (LAT.ADJ$right (reflection ?b ?c)) (extent ?b))
                     (extent ?c))
                  (= (REL.FTN$composition (LAT.ADJ$right (reflection ?b ?c)) (intent ?b))
                     (REL.FTN$composition (extent ?c) (left-derivation c?)))
                  (= (REL.FTN$composition (LAT.ADJ$left (reflection ?b ?c)) (extent ?c))
                     (REL.FTN$composition (extent ?b) (instance-closure ?b)))
                  (= (REL.FTN$composition (LAT.ADJ$left (reflection ?b ?c)) (intent ?c))
                     (REL.FTN$composition (extent ?b) (left-derivation ?b))))))
```

- For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, we can restrict the instance-generation and type-generation to elements, thus defining an instance embedding function $\iota_{\boldsymbol{A}} : \mathit{inst}(\boldsymbol{A}) \to \mathit{latt}(\boldsymbol{A})$ and a type embedding function $\tau_{\boldsymbol{A}} : \mathit{typ}(\boldsymbol{A}) \to \mathit{latt}(\boldsymbol{A})$.

```
(16) (CNG$function instance-embedding)
     (CNG$signature instance-embedding CLS$classification SET.FTN$function)
     (forall (?a (CLS$classification ?a))
         (and (= (SET.FTN$source (instance-embedding ?a)) (CLS$instance ?a))
              (= (SET.FTN$target (instance-embedding ?a)) (concept ?a))
              (= (instance-embedding ?a)
                 (SET.FTN$composition
                     (SET.FTN$singleton (CLS$instance ?a))
                     (instance-generation ?a))))))
```

```
(17) (CNG$function type-embedding)
     (CNG$signature type-embedding CLS$classification SET.FTN$function)
     (forall (?a (CLS$classification ?a))
         (and (= (SET.FTN$source (type-embedding ?a)) (CLS$type ?a))
              (= (SET.FTN$target (type-embedding ?a)) (concept ?a))
              (= (type-embedding ?a)
                 (SET.FTN$composition
                     (SET.FTN$singleton (CLS$type ?a))
                     (type-generation ?a))))))
```

- By means of these two embedding function, we can prove that the notions of extent and intent for classifications extends to the notions of extent and intent for concept lattices.

  $\mathit{extent}_{\boldsymbol{A}}$ (for classifications) = $\tau_{\boldsymbol{A}} \cdot \mathit{extent}_{\boldsymbol{A}}$ (for concept lattices)

  $\mathit{intent}_{\boldsymbol{A}}$ (for classifications) = $\iota_{\boldsymbol{A}} \cdot \mathit{intent}_{\boldsymbol{A}}$ (for concept lattices)

  For clarity, we state these identities in an external namespace.

```
     (forall (?a (CLS$classification ?a))
         (and (= (CLS$extent ?a)
                 (SET.FTN$composition (CLS.CL$type-embedding ?a) (CLS.CL$extent ?a)))
              (= (CLS$intent ?a)
                 (SET.FTN$composition (CLS.CL$instance-embedding ?a) (CLS.CL$intent ?a)))
```

- Concepts in $\iota_{\boldsymbol{A}}(\mathit{inst}(\boldsymbol{A}))$ are called *instance concepts*, whereas concepts in $\tau_{\boldsymbol{A}}(\mathit{typ}(\boldsymbol{A}))$ are called *type concepts*.

```
(18) (CNG$function instance-concept)
     (CNG$signature instance-concept CLS$classification CLS$class)
```

```
        (forall (?a (CLS$classification ?a))
            (and (SET$subclass (instance-concept ?a) (concept ?a))
                 (= (instance-concept ?a)
                    (SET.FTN$image (instance-embedding ?a)))))

(19) (CNG$function type-concept)
     (CNG$signature type-concept CLS$classification CLS$class)
     (forall (?a (CLS$classification ?a))
         (and (SET$subclass (type-concept ?a) (concept ?a))
              (= (type-concept ?a)
                 (SET.FTN$image (type-embedding ?a)))))
```

- Because of the resolutions

  $c = \sqcap_{i \in extent\mathcal{A}(c)} \iota_{\mathcal{A}}(i) = \sqcap_{t \in intent\mathcal{A}(c)} \tau_{\mathcal{A}}(t)$

  for any concept $c \in concept(\mathcal{A})$, these functions satisfy the following conditions.

  - The image $\iota_{\mathcal{A}}(inst(\mathcal{A}))$ is join-dense in $latt(\mathcal{A})$.
  - The image $\tau_{\mathcal{A}}(typ(\mathcal{A}))$ is meet-dense in $latt(\mathcal{A})$.
  - The classification incidence can be expressed in terms of embeddings and lattice order:

    $i \models_{\mathcal{A}} t$ iff $\iota_{\mathcal{A}}(i) \leq_{\mathcal{A}} \tau_{\mathcal{A}}(t)$.

```
(forall (?a (CLS$classification ?a))
    (and ((ORD$join-dense (complete-lattice ?a)) (instance-concept ?a))
         ((ORD$meet-dense (complete-lattice ?a)) (type-concept ?a))
         (forall (?i ((instance ?a) ?i) ?t ((type ?a) ?t))
             (<=> ((CLS$incidence ?a) [?i ?t])
                  ((ORD$extent (LAT$underlying (complete-lattice ?a)))
                      [((instance-embedding ?a) ?i) ((type-embedding ?a) ?t)])))))
```

- By the join-density property stated above, any concept in the lattice can be expressed as the join of a subset of join concepts.

```
(forall (?a (CLS$classification ?a))
        ?c ((concept ?a) ?c))
    (and (exists (?X (SET$subclass ?X (instance-concept ?a)))
             (= ?c ((join ?a) ?X)))
         (exists (?Y (SET$subclass ?Y (type-concept ?a)))
             (= ?c ((meet ?a) ?Y)))))
```

- For any classification $\mathcal{A} = \langle inst(\mathcal{A}), typ(\mathcal{A}), \models_{\mathcal{A}} \rangle$, there are two binary relations that correspond to the instance and type embedding functions.

  Define the instance embedding classification $\iota_{\mathcal{A}} : inst(\mathcal{A}) \to concept(\mathcal{A})$ called *iota* as follows: for every instance $a \in inst(\mathcal{A})$ and every formal concept $\boldsymbol{a} \in concept(\mathcal{A})$, the incidence relationship $a \iota_{\mathcal{A}} \boldsymbol{a}$ holds when a is in the extent of $\boldsymbol{a}$; that is, $a \in ext_{\mathcal{A}}(\boldsymbol{a})$. As a relation, this classification is closed on the right with respect to lattice order. The instance embedding relation can be defined in terms of the instance embedding function as follows:

  $a \iota_{\mathcal{A}} \boldsymbol{a}$ when $\iota_{\mathcal{A}}(a) \leq_{\mathcal{A}} \boldsymbol{a}$, for $a \in inst(\mathcal{A})$ and $\boldsymbol{a} \in concept(\mathcal{A})$.

  This is an application of the right operator for a preorder that maps functions to binary relations.

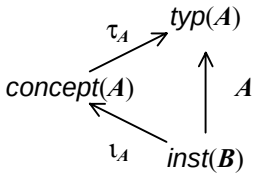


Figure 4: ***iota*** **&** ***tau***

  Dually, define the type embedding classification $\tau_{\mathcal{A}} : concept(\mathcal{A}) \to typ(\mathcal{A})$ called *tau* as follows: for every formal concept $\boldsymbol{a} \in concept(\mathcal{A})$ and every type $\alpha \in typ(\mathcal{A})$, the incidence relationship $\boldsymbol{a} \tau_{\mathcal{A}} \alpha$ holds when α is in the intent of $\boldsymbol{a}$; that is, $\alpha \in int_{\mathcal{A}}(\boldsymbol{a})$. As a relation, this classification is closed on the left with respect to lattice order. The type embedding relation can be defined in terms of the type embedding function as follows:

  $\boldsymbol{a} \tau_{\mathcal{A}} \alpha$ when $\boldsymbol{a} \leq_{\mathcal{A}} \tau_{\mathcal{A}}(\alpha)$, for $\boldsymbol{a} \in concept(\mathcal{A})$ and $\alpha \in typ(\mathcal{A})$.

  This is an application of the left operator for a preorder that maps functions to binary relations.

Note the direction of the type embedding relation.

```
(20) (CNG$function iota)
     (CNG$signature iota CLS$classification CLS$classification)
     (forall (?a (CLS$classification ?a))
         (and (= (REL$instance (iota ?a)) (CLS$instance ?a))
              (= (REL$type (iota ?a)) (concept ?a))
              (= (iota ?a)
                 ((SET.FTN$right (concept-order ?a)) (instance-embedding ?a)))))

(21) (CNG$function tau)
     (CNG$signature tau CLS$classification CLS$classification)
     (forall (?a (CLS$classification ?a))
         (and (= (REL$instance (tau ?a)) (concept ?a))
              (= (REL$type (tau ?a)) (CLS$type ?a))
              (= (tau ?a)
                 ((SET.FTN$left (concept-order ?a)) (type-embedding ?a)))))
```

- The instance embedding function can be defined in terms of the instant embedding relation as follows:

  $\iota_A(a) = \sqcap_A(a\iota_A)$, for $a \in \mathit{inst}(\boldsymbol{A})$.

  The type embedding function can be defined in terms of the type embedding relation as follows:

  $\tau_A(\alpha) = \sqcup_A(\tau_A\alpha)$, for $\alpha \in \mathit{typ}(\boldsymbol{A})$.

  However, since we have already defined these functions by other means, these facts are expressed as theorems.

```
(forall (?a (CLS$classification ?a))
         ?x ((CLS$instance ?a) ?x))
    (= ((instance-embedding ?a) ?x)
       ((meet ?a) ((REL$fiber12 (iota ?a)) ?x))))

(forall (?a (CLS$classification ?a))
         ?y ((CLS$type ?a) ?y))
    (= ((type-embedding ?a) ?y)
       ((join ?a) ((REL$fiber21 (tau ?a)) ?y))))
```

- Intent induces a preorder on the instance class $\mathit{inst}(\boldsymbol{A})$ defined by $i_1 \leq_A i_2$ when $\iota_A(i_1) \leq_A \iota_A(i_2)$ or $i_1' \supseteq i_2'$. Dually, extent induces a preorder on the type class $\mathit{typ}(\boldsymbol{A})$ defined by $t_1 \leq_A t_2$ when $\tau_A(t_1) \leq_A \tau_A(t_2)$ or $t_1' \subseteq t_2'$.

```
(22) (CNG$function instance-order)
     (CNG$signature instance-order CLS$classification ORD$preorder)
     (forall (?a (CLS$classification ?a))
         (and (= (ORD$class (instance-order ?a)) (instance ?a))
              (forall (?i1 ((instance ?a) ?i1) (?i2 ((instance ?a) ?i2))
              (<=> ((ORD$extent (instance-order ?a)) [?i1 ?i2])
                   ((ORD$extent (concept-order ?a))
                       [((instance-embedding ?a) ?i1)
                        ((instance-embedding ?a) ?i2)])))))

(23) (CNG$function type-order)
     (CNG$signature type-order CLS$classification ORD$preorder)
     (forall (?a (CLS$classification ?a))
         (and (= (ORD$class (type-order ?a)) (type ?a))
              (forall (?t1 ((type ?a) ?t1) (?t2 ((type ?a) ?t2))
              (<=> ((ORD$extent (type-order ?a)) [?t1 ?t2])
                   ((ORD$extent (concept-order ?a))
                       [((type-embedding ?a) ?t1)
                        ((type-embedding ?a) ?t2)])))))
```

- Part of the "fundamental theorem" of Formal Concept Analysis states that every classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \vDash_A \rangle$ has an associated concept lattice $\mathit{cl}(\boldsymbol{A}) = \langle \mathit{latt}(\boldsymbol{A}), \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \iota_A, \tau_A \rangle$.

```
(24) (CNG$function concept-lattice)
     (CNG$signature concept-lattice CLS$classification CL$concept-lattice)
     (forall (?a (CLS$classification ?a))
         (and (= (CL$complete-lattice (concept-lattice ?a)) (complete-lattice ?a))
              (= (CL$instance (concept-lattice ?a)) (CLS$instance ?a))
```

```
(= (CL$type (concept-lattice ?a)) (CLS$type ?a))
(= (CL$instance-embedding (concept-lattice ?a)) (instance-embedding ?a))
(= (CL$type-embedding (concept-lattice ?a)) (type-embedding ?a))))
```

- The following fact of abstract Formal Concept Analysis, stated in terms of the embedding functions above, is straightforward to prove:

  $$\boldsymbol{A} = \iota_{\boldsymbol{A}} \circ \leq_{\boldsymbol{A}} \circ \tau_{\boldsymbol{A}}.$$

  This says that any classification (viewed as a binary relation) is identical to the composition of the instance-embedding relation followed by the lattice order of the classification followed by the type embedding relation.

```
(forall (?a (CLS$classification ?a))
    (= ?a
       (REL$composition (REL$composition (iota ?a) (concept-order ?a)) (tau ?a))))
```

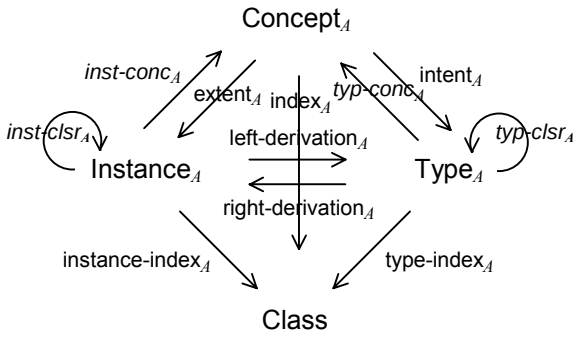


Diagram 5: Core Conglomerates and Functions for Fibers

Conceptual Fibers

CLS.FIB

This section defines fibers along the underlying classification function or compositions with this function (see Diagram 6).

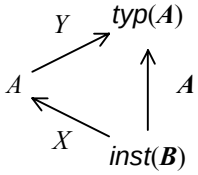


Figure 4: *A*-instances & *A*-types

- For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, a (*collective*) ***A**-instance* indexed by a class A is a binary relation $X \subseteq \mathit{inst}(\boldsymbol{A}) \times A$. For a fixed classification $\boldsymbol{A}$, the collection of all $\boldsymbol{A}$-instances is denoted '`(instance ?a)`'. Dually, a (*collective*) ***A**-type* indexed by a class A is a binary relation $Y \subseteq A \times \mathit{typ}(\boldsymbol{A})$. For a fixed classification $\boldsymbol{A}$, the collection of all $\boldsymbol{A}$-types is denoted '`(type ?a)`'.

```
(1) (KIF$function instance)
    (KIF$signature instance CLS$classification CNG$conglomerate)
    (forall (?a (CLS$classification ?a)
        (and (CNG$subconglomerate (instance ?a) REL$relation)
             (forall (?x (REL$relation ?x))
                 (<=> ((instance ?a) ?x)
                      (= (REL$source ?x) (CLS$instance ?a))))))

(2) (KIF$function instance-index)
    (KIF$signature instance-index CLS$classification CNG$function)
    (forall (?a (CLS$classification ?a))
        (and (CNG$signature (instance-index ?a) (instance ?a) SET$class)
             (forall (?x ((instance ?a) ?x))
                 (= ((instance-index ?a) ?x) (REL$target ?x)))))

(3) (KIF$function type)
    (KIF$signature type CLS$classification CNG$conglomerate)
    (forall (?a (CLS$classification ?a)
        (and (CNG$subconglomerate (type ?a) REL$relation)
             (forall (?y (REL$relation ?y))
                 (<=> ((type ?a) ?y)
                      (= (REL$target ?y) (CLS$type ?a))))))

(4) (KIF$function type-index)
    (KIF$signature type-index CLS$classification CNG$function)
    (forall (?a (CLS$classification ?a))
        (and (CNG$signature (type-index ?a) (type ?a) SET$class)
             (forall (?y ((type ?a) ?y))
                 (= ((type-index ?a) ?y) (REL$source ?y)))))
```

- Two derivation operators in this fibered setting correspond to the two relational residuation operators. For any large classification $\boldsymbol{A}$ there are two senses of *derivation* corresponding to the two senses of relational residuation. Derivation preserves indices. Left derivation maps an instance to the type on which it is "universally incident," and dually, right derivation maps a type to the instance which is "universally incident" on it: for all instances $X \subseteq \mathit{inst}(\boldsymbol{A}) \times A$ and all types $Y \subseteq A \times \mathit{typ}(\boldsymbol{A})$,

$X \mapsto X' = X \backslash \boldsymbol{A}$ and $Y \mapsto Y' = \boldsymbol{A} / Y$.

```
(5) (KIF$function left-derivation)
    (KIF$signature left-derivation CLS$classification CNG$function)
    (forall (?a (CLS$classification ?a))
        (and (CNG$signature (left-derivation ?a) (instance ?a) (type ?a))
             (forall (?x ((instance ?a) ?x))
                 (= ((left-derivation ?a) ?x)
                    (REL$left-residuation ?x ?a)))))

(6) (CNG$function right-derivation)
    (CNG$signature right-derivation CLS$classification SET.FTN$function)
    (forall (?a (CLS$classification ?a))
        (and (CNG$signature (right-derivation ?a) (type ?a) (instance ?a))
             (forall (?y ((type ?a) ?y))
                 (= ((right-derivation ?a) ?y)
                    (REL$right-residuation ?y ?a)))))
```

- A simple fundamental result is the following equivalence. For all instances $X \subseteq \mathit{inst}(\boldsymbol{A}) \times A$ and all types $Y \subseteq A \times \mathit{typ}(\boldsymbol{A})$

  $Y \subseteq X \backslash \boldsymbol{A}$ in $\mathit{REL}[A, \mathit{typ}(\boldsymbol{A})]$ iff $X \circ Y \subseteq \boldsymbol{A}$ iff $X \subseteq \boldsymbol{A} / Y$ in $\mathit{REL}[\mathit{inst}(\boldsymbol{A}), A]$.

  This describes a Galois connection between left derivation

  $(\text{-}) \backslash \boldsymbol{A} : \mathit{REL}[\mathit{inst}(\boldsymbol{A}), A] \to (\mathit{REL}[A, \mathit{typ}(\boldsymbol{A})])^{\mathrm{op}}$

  and right derivation

  $\boldsymbol{A} / (\text{-}) : (\mathit{REL}[A, \mathit{typ}(\boldsymbol{A})])^{\mathrm{op}} \to \mathit{REL}[\mathit{inst}(\boldsymbol{A}), A]$.

  The first two facts assert (contravariant) monotonicity of derivation. The last fact asserts the adjointness condition.

```
    (forall (?x1 ((instance ?a) ?x1)
             ?x2 ((instance ?a) ?x2)
             (= ((instance-index ?a) ?x1)
                ((instance-index ?a) ?x2)))
        (=> (REL$subrelation ?x1 ?x2)
            (SET$subrelation ((left-derivation ?a) ?x2) ((left-derivation ?a) ?x1))))

    (forall (?y1 ((type ?a) ?y1)
             ?y2 ((type ?a) ?y2)
             (= ((type-index ?a) ?y1)
                ((type-index ?a) ?y2)))
        (=> (REL$subrelation ?y2 ?y1)
            (REL$subrelation ((right-derivation ?a) ?y1) ((right-derivation ?a) ?y2))))

    (forall (?x ((instance ?a) ?x)
             ?y ((type ?a) ?y)
             (= ((instance-index ?a) ?x)
                ((type-index ?a) ?y)))
        (<=> (REL$subrelation ?y ((left-derivation ?a) ?x))
             (REL$subrelation ?x ((right-derivation ?a) ?y))))
```

- The composition of derivations gives two senses of closure operator.

```
(7) (KIF$function instance-closure)
    (KIF$signature instance-closure CLS$classification CNG$function)
    (forall (?a (CLS$classification ?a))
        (and (CNG$signature (instance-closure ?a) (instance ?a) (instance ?a))
             (forall (?x ((instance ?a) ?x))
                 (and (= (instance-index ((instance-closure ?a) ?x))
                         (instance-index ?x))
                      (= ((instance-closure ?a) ?x)
                         ((right-derivation ?a) ((left-derivation ?a) ?x)))))))

(8) (CNG$function type-closure)
     (CNG$signature type-closure CLS$classification CNG$function)
        (and (CNG$signature (type-closure ?a) (type ?a) (type ?a))
```

```
(forall (?y ((type ?a) ?y))
    (and (= (type-index ((type-closure ?a) ?y))
            (type-index ?y))
         (= ((type-closure ?a) ?y)
            ((left-derivation ?a) ((right-derivation ?a) ?y)))))))
```

- The following (easily proven) results confirm that these are closure operators.

  $X \subseteq X''$ and $X' = X'''$ for all instances $X \subseteq \mathit{inst}(\boldsymbol{A}) \times A$.

  $Y \subseteq Y''$ and $Y' = Y'''$ for all types $Y \subseteq A \times \mathit{typ}(\boldsymbol{A})$.

```
(forall (?x ((instance ?a) ?x))
    (and (REL$subrelation ?x ((instance-closure ?a) ?x))
         (= ((left-derivation ?a) ?x)
            ((left-derivation ?a) ((instance-closure ?a) ?x)))))

(forall (?y ((type ?a) ?y))
    (and (REL$subrelation ?y ((type-closure ?a) ?y))
         (= ((right-derivation ?a) ?y)
            ((right-derivation ?a) ((type-closure ?a) ?y)))))
```

- For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, a (*collective*) $\boldsymbol{A}$-*concept* $\boldsymbol{C} = \langle \mathit{ind}(\boldsymbol{C}), \mathit{ext}(\boldsymbol{C}), \mathit{int}(\boldsymbol{C}) \rangle$, consists of an *extent* $\boldsymbol{A}$-instance $\mathit{ext}(\boldsymbol{C}) : \mathit{inst}(\boldsymbol{A}) \to \mathit{ind}(\boldsymbol{A})$ indexed by $\mathit{ind}(\boldsymbol{A})$, and an *intent* $\boldsymbol{A}$-type $\mathit{int}(\boldsymbol{C}) : \mathit{ind}(\boldsymbol{A}) \to \mathit{typ}(\boldsymbol{A})$ indexed by $\mathit{ind}(\boldsymbol{A})$, which satisfy the following closure conditions:

  $\mathit{ext}(\boldsymbol{C}) = \boldsymbol{A}/\mathit{int}(\boldsymbol{C})$ and $\mathit{int}(\boldsymbol{C}) = \mathit{ext}(\boldsymbol{C}) \backslash \boldsymbol{A}$.

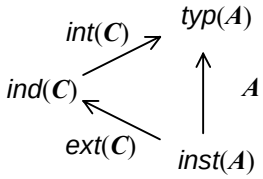


Figure 5: $\boldsymbol{A}$-Concept

  The collection of all $\boldsymbol{A}$-concepts is denoted by '`(concept ?a)`'.

```
(9) (KIF$function concept)
    (KIF$signature concept CLS$classification CNG$conglomerate)

(10) (KIF$function extent)
     (KIF$signature extent CLS$classification CNG$function)
     (forall (?a (CLS$classification ?a))
         (CNG$signature (extent ?a) (concept ?a) (instance ?a))

(11) (KIF$function intent)
     (KIF$signature intent CLS$classification CNG$function)
     (forall (?a (CLS$classification ?a))
         (CNG$signature (intent ?a) (concept ?a) (type ?a))

(14) (KIF$function index)
     (KIF$signature index CLS$classification CNG$function)
     (forall (?a (CLS$classification ?a))
         (and (CNG$signature (index ?a) (concept ?a) SET$class)
              (forall (?c ((concept ?a) ?c))
                  (and (= ((index ?a) ?c)
                          ((instance-index ?a) ((extent ?a) ?c)))
                       (= ((index ?a) ?c)
                          ((type-index ?a) ((intent ?a) ?c)))
                       (= ((left-derivation ?a) ((extent ?a) ?c))
                          ((intent ?a) ?c))
                       (= ((right-derivation ?a) ((intent ?a) ?c))
                          ((extent ?a) ?c))))))
```

- There are surjective generator functions, called instance-generation and type-generation, that map instances and types to their generated concepts. For all instances $X \subseteq \mathit{inst}(\boldsymbol{A}) \times A$ and types $Y \subseteq A \times \mathit{typ}(\boldsymbol{A})$

  $X \mapsto \langle X'', X' \rangle$ "instance-generation"

  $Y \mapsto \langle Y', Y'' \rangle$ "type-generation".

  – The composition of instance-generation and intent equals left-derivation, and the composition of type-generation and extent equals right-derivation.

- The composition of instance-generation and extent equals instance-closure, and the composition of type-generation and intent equals type-closure.
- The composition of extent and instance-generation is identity, and the composition of intent and type-generation is identity.

These conditions define generation, and also insure that all concepts are captured in the conglomerate '`(concept ?a)`'. Concepts are determined by either their extents or their intents – this is represented by the fact that the extent and intent functions are injective.

```
(15) (KIF$function instance-generation)
     (KIF$signature instance-generation CLS$classification CNG$function)
     (forall (?a (CLS$classification ?a))
         (and (CNG$signature (instance-generation ?a) (instance ?a) (concept ?a))
              (forall (?x ((instance ?a) ?x))
                  (and (= ((index ?a) ((instance-generation ?a) ?x))
                          ((instance-index ?a) ?x))
                       (= ((intent ?a) ((instance-generation ?a) ?x))
                          ((left-derivation ?a) ?x))
                       (= ((extent ?a) ((instance-generation ?a) ?x))
                          ((instance-closure ?a) ?x))))
              (forall (?c ((concept ?a) ?c))
                  (= ((instance-generation ?a) ((extent ?a) ?c)) ?c))))

(16) (KIF$function type-generation)
     (KIF$signature type-generation CLS$classification CNG$function)
     (forall (?a (CLS$classification ?a))
         (and (CNG$signature (type-generation ?a) (type ?a) (concept ?a))
              (forall (?y ((type ?a) ?y))
                  (and (= ((index ?a) ((type-generation ?a) ?y))
                          ((type-index ?a) ?y))
                       (= ((extent ?a) ((type-generation ?a) ?y))
                          ((right-derivation ?a) ?y))
                       (= ((intent ?a) ((type-generation ?a) ?y))
                          ((type-closure ?a) ?y))))
              (forall (?c ((concept ?a) ?c))
                  (= ((type-generation ?a) ((intent ?a) ?c)) ?c))))
```

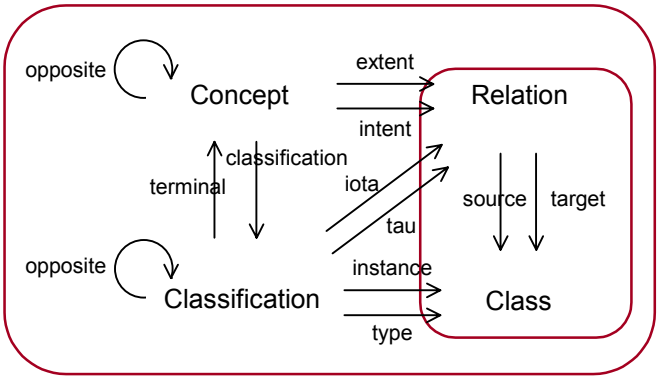


Diagram 6: Core Conglomerates and Functions for Concepts

Collective Concepts

`CLS.CONC`

o Here we define collective concepts that internally include their underlying classifications. A formal (collective) concept $\boldsymbol{C} = \langle cls(\boldsymbol{C}), ind(\boldsymbol{C}), ext(\boldsymbol{C}), int(\boldsymbol{C})\rangle$ consists of an underlying *classification* $cls(\boldsymbol{C})$, an *index*ing class $ind(\boldsymbol{C})$, an *extent* relation $ext(\boldsymbol{C})$ and an *intent* relation $int(\boldsymbol{C})$. These components satisfy the equivalent closure conditions:

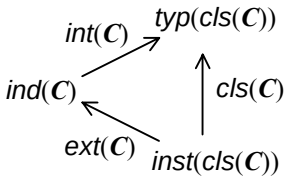


Figure 6: Collective Concept

$$ext(\boldsymbol{C}) = cls(\boldsymbol{C})/int(\boldsymbol{C}) \text{ and } int(\boldsymbol{C}) = ext(\boldsymbol{C})\backslash cls(\boldsymbol{C}).$$

The conglomeration of all concepts is the disjoint union of all the conceptual fibers '`(CLS.FIB$concept ?a)`' as $\boldsymbol{A}$ ranges over the conglomerate of all large classifications.

Axiom (1) specifies the (collective) concept conglomerate. Axiom (2) specifies the underlying classification function. This function is part of a fibration. The index function is specified in axiom (3). When this maps to unity (the unit class), the concept is a point. Axiom (6) expresses the closure conditions. Axiom (7) expresses the disjoint union property.

```
(1) (CNG$conglomerate concept)

(2) (CNG$function classification)
    (CNG$signature classification concept CLS$classification)

(3) (CNG$function index)
    (CNG$signature index concept SET$class)

(4) (CNG$function extent)
    (CNG$signature extent concept REL$relation)
    (forall (?c (concept ?c))
        (and (= (REL$source (extent ?c))
                (CLS$instance (classification ?c)))
             (= (REL$target (extent ?c))
                (index ?c))))

(5) (CNG$function intent)
    (CNG$signature intent concept REL$relation)
    (forall (?c (concept ?c))
        (and (= (REL$source (intent ?c))
                (index ?c))
             (= (REL$target (intent ?c))
                (CLS$type (classification ?c)))))

(6) (forall (?c (concept ?c))
        (and (= (extent ?c)
```

```
                (REL$right-residuation (intent ?c) (classification ?c)))
             (= (intent ?c)
                (REL$left-residuation (extent ?c) (classification ?c)))))

(7) (forall (?c)
        (<=>  (concept ?c)
              (exists (?cf ((CLS.FIB$concept (classfication ?c)) ?cf))
                  (and (= (index ?c)
                          ((CLS.FIB$index (classfication ?c)) ?cf))
                       (= (extent ?c)
                          ((CLS.FIB$extent (classfication ?c)) ?cf))
                       (= (intent ?c)
                          ((CLS.FIB$intent (classfication ?c)) ?cf))))))
```

- For any concept $\boldsymbol{C} = \langle \mathit{cls}(\boldsymbol{C}), \mathit{ind}(\boldsymbol{C}), \mathit{ext}(\boldsymbol{C}), \mathit{int}(\boldsymbol{C}) \rangle$, the *opposite* or *dual* of $\boldsymbol{C}$ is the concept $\boldsymbol{C}^{\perp} = \langle \mathit{cls}(\boldsymbol{C})^{\perp}, \mathit{ind}(\boldsymbol{C}), \mathit{int}(\boldsymbol{C}), \mathit{ext}(\boldsymbol{C}) \rangle$, whose classification is the opposite of the classification of $\boldsymbol{C}$, whose index is the same as the index of $\boldsymbol{C}$, whose extent is the intent of $\boldsymbol{C}$, and whose intent is the extent of $\boldsymbol{C}$. Axiom (8) specifies the opposite operator on classifications.

```
(8) (CNG$function opposite)
    (CNG$signature opposite concept concept)
    (forall (?c (concept ?c))
        (and (= (classification (opposite ?c)) (CLS$opposite (classification ?c)))
             (= (index (opposite ?c)) (index ?c))
             (= (extent (opposite ?c)) (intent ?c))
             (= (intent (opposite ?c)) (extent ?c))))
```

- For any two collective concepts $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ over the same classification $\mathit{cls}(\boldsymbol{C}_1) = \boldsymbol{A} = \mathit{cls}(\boldsymbol{C}_2)$, a *two cell* $f : \boldsymbol{C}_1 \Rightarrow \boldsymbol{C}_2$ from $\boldsymbol{C}_1$ to $\boldsymbol{C}_2$ is a function $f : \mathit{ind}(\boldsymbol{C}_1) \to \mathit{ind}(\boldsymbol{C}_2)$ between index classes, which satisfies the following conditions:

$$\mathit{ext}(\boldsymbol{C}_1) = \mathit{ext}(\boldsymbol{C}_2) \circ f^{\mathrm{op}} = \mathit{ext}(\boldsymbol{C}_2) / f$$

$$\mathit{int}(\boldsymbol{C}_1) = f \circ \mathit{int}(\boldsymbol{C}_2) = f^{\mathrm{op}} \setminus \mathit{int}(\boldsymbol{C}_2)$$

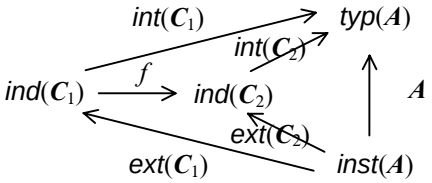


Figure 6: 2-cell

```
(9) (CNG$conglomerate two-cell)

(10) (CNG$function source)
     (CNG$signature source two-cell concept)

(11) (CNG$function target)
     (CNG$signature target two-cell concept)

(12) (CNG$function function)
     (CNG$signature function two-cell SET.FTN$function)

     (forall (?f (two-cell ?f))
         (and (= (classification (source ?f))
                 (classification (target ?f)))
              (= (index (source ?f))
                 (SET.FTN$source (function ?f)))
              (= (index (target ?f))
                 (SET.FTN$target (function ?f)))
              (= (SET.FTN$composition
                     (extent (target ?f))
                     (REL$opposite (SET.FTN$fn2rel (function ?f))))
                 (extent (source ?f)))
              (= (SET.FTN$composition
                     (SET.FTN$fn2rel (function ?f))
                     (intent (target ?f))
                 (intent (source ?f))))))
```

- The closure conditions $\iota_{\boldsymbol{A}} = \models_{\boldsymbol{A}} / \tau_{\boldsymbol{A}}$ and $\tau_{\boldsymbol{A}} = \iota_{\boldsymbol{A}} \backslash \models_{\boldsymbol{A}}$ hold between the instance embedding relation $\iota_{\boldsymbol{A}} : \mathit{inst}(\boldsymbol{A}) \to \mathit{concept}(\boldsymbol{A})$ and the type embedding relation $\tau_{\boldsymbol{A}} : \mathit{concept}(\boldsymbol{A}) \to \mathit{typ}(\boldsymbol{A})$. So, for any classification $\boldsymbol{A}$, a basic example of a collective concept is $\mathit{terminal}(\boldsymbol{A}) = \langle \boldsymbol{A}, \mathit{concept}(\boldsymbol{A}), \iota_{\boldsymbol{A}}, \tau_{\boldsymbol{A}} \rangle$ with $\iota_{\boldsymbol{A}}$ as collective extent, $\tau_{\boldsymbol{A}}$ as collective intent, and the concept class $\mathit{concept}(\boldsymbol{A})$ as index.

```
(13) (CNG$function terminal)
     (CNG$signature terminal CLS$classification concept)
     (forall (?a (CLS$classification ?a))
         (and (= (classification (terminal ?a)) ?a)
              (= (index (terminal ?a)) (CLS.CL$concept ?a))
              (= (extent (terminal ?a)) (iota ?a))
              (= (intent (terminal ?a)) (tau ?a))))
```

- Any collective concept $\boldsymbol{C} = \langle \boldsymbol{A}, A, X, Y \rangle = \langle \mathit{cls}(\boldsymbol{C}), \mathit{ind}(\boldsymbol{C}), \mathit{ext}(\boldsymbol{C}), \mathit{int}(\boldsymbol{C}) \rangle$ induces a unique mediating function $\mu_{\boldsymbol{C}} : \mathit{ind}(\boldsymbol{C}) \to \mathit{cl}(\mathit{cls}(\boldsymbol{C}))$ that satisfies the following constraints.

$$\mathit{ext}(\boldsymbol{C}) = \iota_{\mathit{cls}(\boldsymbol{C})} \circ \mu_{\boldsymbol{C}}{}^{\mathrm{op}} = \iota_{\mathit{cls}(\boldsymbol{C})} / \mu_{\boldsymbol{C}}$$

$$\mathit{int}(\boldsymbol{C}) = \mu_{\boldsymbol{C}} \circ \tau_{\mathit{cls}(\boldsymbol{C})} = \mu_{\boldsymbol{C}}{}^{\mathrm{op}} \setminus \tau_{\mathit{cls}(\boldsymbol{C})}$$

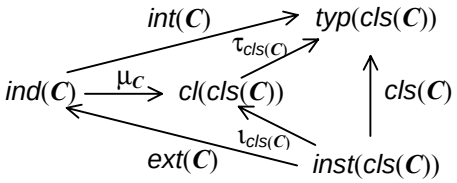


Figure 6: Mediating Function

The definition $\mu_{\boldsymbol{C}}(a) = \langle Xa, aY \rangle$ is well-defined, since the closure conditions are equivalent to the (pointwise) fact that $Xa = (aY)'$ and $aY = (Xa)'$; that is, that $\langle Xa, aY \rangle \in \mathit{latt}(\boldsymbol{A})$. Conversely, for any classification $\boldsymbol{A}$ and for any function $f : \mathit{ind}(\boldsymbol{C}) \to \mathit{cl}(\boldsymbol{A})$, the quadruple $\langle \boldsymbol{A}, \mathit{concept}(\boldsymbol{A}), \iota_{\mathit{cls}(\boldsymbol{C})} \circ f^{\mathrm{op}}, f \circ \tau_{\mathit{cls}(\boldsymbol{C})} \rangle$ is an collective concept. So the CNG function '`(mediator-function ?c)`' defined below is a bijection.

```
(14) (CNG$function mediator-function)
     (CNG$signature mediator-function concept SET.FTN$function)
     (forall (?c (concept ?c))
         (and (= (SET.FTN$source (mediator-function ?c))
                 (index ?c))
              (= (SET.FTN$target (mediator-function ?c))
                 (CLS.CL$concept (classification ?c)))
              (= (SET.FTN$composition
                     (mediator-function ?c)
                     (CLS.CL$extent (classification ?c)))
                 (SET.FTN$fiber21 (extent ?c)))
              (= (SET.FTN$composition
                     (mediator-function ?c)
                     (CLS.CL$intent (classification ?c)))
                 (SET.FTN$fiber12 (intent ?c)))))
```

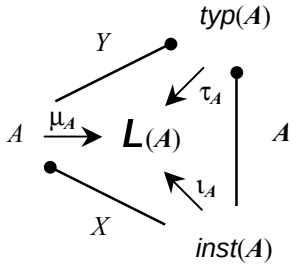
- For any classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, there is a special (collective) concept that is in a sense is the largest concept over that classification. Its extent relation $\iota_{\boldsymbol{A}} \subseteq \mathit{inst}(\boldsymbol{A}) \times \mathit{concept}(\boldsymbol{A})$ is the instance embedding relation *iota*, and its intent relation $\tau_{\boldsymbol{A}} \subseteq \mathit{concept}(\boldsymbol{A}) \times \mathit{typ}(\boldsymbol{A})$ is the type embedding relation *tau*.

- There are bijections between subsets of instances/types and collective instances/types. Any subclass of instances $X \subseteq \mathit{inst}(\boldsymbol{A})$ is a (has an associated) collective $\boldsymbol{A}$-instance $\mathit{inst\text{-}dist}(X) \subseteq \mathit{inst}(\boldsymbol{A}) \times 1$ indexed by the unit class 1. A similar statement holds for types.

```
(KIF$function instance-distribution)
(KIF$signature instance-distribution CLS$classification CNG$function)
(forall (?a (CLS$classification ?a))
    (and (CNG$signature (instance-distribution ?a)
             (SET$power (CLS$instance ?a)) (instance ?a))
         (forall (?X (SET$subclass ?X (CLS$instance ?a)))
             (and (= ((instance-index ?a) ((instance-distribution ?a) ?X))
                     SET.LIM$unit)
                  (forall (?x ((CLS$instance ?a) ?x))
                      (<=> ((REL$extent ((instance-distribution ?a) ?X)) [?x 0])
                           (?X ?x)))))))

(KIF$function type-distribution)
(KIF$signature type-distribution CLS$classification CNG$function)
(forall (?a (CLS$classification ?a))
    (and (CNG$signature (type-distribution ?a)
             (SET$power (CLS$type ?a)) (type ?a))
         (forall (?Y (SET$subclass ?Y (CLS$type ?a)))
             (and (= ((type-index ?a) ((type-distribution ?a) ?Y))
```

```
    SET.LIM$unit)
(forall (?y ((CLS$type ?a) ?y))
    (<=> ((REL$extent ((type-distribution ?a) ?X)) [0 ?y])
         (?Y ?y)))))))
```

- A collective concept is also called a *concept space*. The indexed or named formal concepts in a concept space are also called *conceptual views*. Conceptual knowledge is represented the following components of a concept space.
  - The three preorders on instances, types and conceptual views.
  - The membership or instantiation relation between instances and conceptual views, whose columns, when regarded as a Boolean matrix, record the *extent* of all the conceptual views.



$\boldsymbol{A}$	*classification*
$\mathit{inst}(\boldsymbol{A})$	*instance preorder*
$\mathit{typ}(\boldsymbol{A})$	*type preorder*
A	*index preorder*
X	*extent*
Y	*intent*
$\mathsf{L}(\boldsymbol{A})$	*concept lattice*

  - The abstraction relation between conceptual views and types, whose rows, when regarded as a Boolean matrix, record the *intent* of all the conceptual views.
  - The *classification* relation between instances and types.

The constraining relation ships between the extent and intent of a concept space can be expressed in three different forms: residuation, inclusion and derivation.

incidence constraints		
residuation	*inclusion*	*derivation*
$Y = X \backslash \boldsymbol{A}$ $X = \boldsymbol{A} / Y$	$\forall_{a \in A,\, t \in \mathit{typ}(\boldsymbol{A})}\, aYt \;\; \mathit{iff} \;\; Xa \subseteq \boldsymbol{A}t$ $\forall_{i \in \mathit{inst}(\boldsymbol{A}),\, a \in A}\, iXa \;\; \mathit{iff} \;\; i\boldsymbol{A} \supseteq aY$	$\forall_{a \in A} \;\; aY = (Xa)'$ $\forall_{a \in A} \;\; Xa = (aY)'$

In order to construct a concept space, we start out by specifying the class A to be a collection of names, with each $a \in A$ representing a subset of types $aY_0 \subseteq \mathit{typ}(\boldsymbol{A})$. We use the two residuation operators, which are a generalized form of the derivation operators of Formal Concept Analysis, in order to define the notion of a *collective formal concept*. We define the extent X to be the right residuation of $\boldsymbol{A}$ along Y_0:

$$X = \boldsymbol{A} / Y_0 = \{(i, a) \mid \forall_{t \in \mathit{typ}(\boldsymbol{A})}\, (aY_0t \Rightarrow i\boldsymbol{A}t)\}.$$

so that $X \circ Y_0 \subseteq \boldsymbol{A}$. We define Y to be the left residuation of $\boldsymbol{A}$ along X:

$$Y = X \backslash \boldsymbol{A} = \{(a, t) \mid \forall_{i \in \mathit{inst}(\boldsymbol{A})}\, (iXa \Rightarrow i\boldsymbol{A}t)\}.$$

From these definitions it is straightforward to show that X is the right residuation, $X = \boldsymbol{A}/Y$, of $\boldsymbol{A}$ along Y: First of all, since $Y_0 \subseteq X \backslash \boldsymbol{A} = Y$, by contravariance of residuation $X = \boldsymbol{A}/Y_0 \supseteq \boldsymbol{A}/Y$; and secondly, since $X \circ Y = X \circ (X \backslash \boldsymbol{A}) \subseteq \boldsymbol{A}$, we have $X \subseteq \boldsymbol{A}/Y$.

Table 2: Conglomerate Functions used to define the Concept Lattice Functor

left derivation : Classification → Function
right derivation : Classification → Function
instance closure : Classification → Function
type closure : Classification → Function
concept : Classification → Class
extent : Classification → Function
intent : Classification → Function
instance concept : Classification → Function
type concept : Classification → Function
concept order : Classification → Partial Order
meet : Classification → Function
join : Classification → Function
complete lattice : Classification → Complete Lattice
adjoint pair : Infomorphism → Adjoint Pair
instance embedding : Classification → Function
type embedding : Classification → Function
concept lattice : Classification → Concept Lattice
concept morphism : Infomorphism → Concept Morphism
instance embedding relation : Classification → Relation
type embedding relation : Classification → Relation

Table 3: Functors and Natural Isomorphisms

$L \circ C = Id_{\mathbf{Classification}}$	`CLS.CL$concept-lattice . CL$classification = Id`
	`CLS.INFO$concept-morphism . CL.MOR$infomorphism = Id`
$C \circ L \cong Id_{\mathbf{Classification}}$	`CL$classification . CLS.CL$concept-lattice ≅ Id`
	`CL.MOR$infomorphism . CLS.INFO$concept-morphism ≅ Id`

Functional Infomorphisms

CLS.INFO

- Classifications are related through (functional) infomorphisms. A (*functional*) *infomorphism* $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ (see Figure 2) is a pair $\boldsymbol{f} = \langle \mathit{inst}(\boldsymbol{f}), \mathit{typ}(\boldsymbol{f}) \rangle$ of oppositely directed functions, $\mathit{inst}(\boldsymbol{f}) : \mathit{inst}(\boldsymbol{B}) \to \mathit{inst}(\boldsymbol{A})$ and $\mathit{typ}(\boldsymbol{f}) : \mathit{typ}(\boldsymbol{A}) \to \mathit{typ}(\boldsymbol{B})$, which satisfy the fundamental property:

$$\mathit{inst}(\boldsymbol{f})(i) \vDash_{\boldsymbol{A}} t \text{ iff } i \vDash_{\boldsymbol{B}} \mathit{typ}(\boldsymbol{f})(t)$$

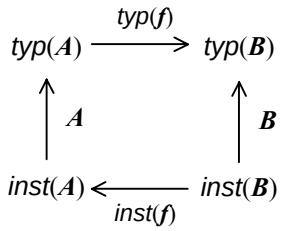


Figure 2: Functional Infomorphism

for all instances $i \in \mathit{inst}(\boldsymbol{B})$ and all types $t \in \mathit{typ}(\boldsymbol{A})$. Note that the relation expressed on the either side of this equivalence is the bond of the relational infomorphism generated by this functional infomorphism. This *relational infomorphism* is defined below in terms of the left relation of the *instance monotonic function* and the right relation of the *type monotonic function*.

The following is a KIF representation for the elements of an infomorphism. Such elements are useful for the definition of an infomorphism. In the same fashion as classifications, infomorphisms are specified by declaration and population. The term '`infomorphism`' specified in axiom (1) allows one to *declare* infomorphisms themselves, and the two terms '`source`' and '`target`' specified in axioms (2–3) allow one to *declare* their associated source (domain) and target (codomain) classifications, respectively. The terms '`instance`' and '`type`' specified in axioms (4–5) resolve infomorphisms into their parts, thus allowing one to *populate* infomorphisms. The Infomorphism namespace is a sub-namespace of the Classification namespace.

```
(1) (CNG$conglomeration infomorphism)

(2) (CNG$function source)
    (CNG$signature source infomorphism CLS$classification)

(3) (CNG$function target)
    (CNG$signature target infomorphism CLS$classification)

(4) (CNG$function instance)
    (CNG$signature instance infomorphism SET.FTN$function)
    (forall (?f (infomorphism ?f))
        (and (= (SET.FTN$source (instance ?f)) (CLS$instance (target ?f)))
             (= (SET.FTN$target (instance ?f)) (CLS$instance (source ?f)))))

(5) (CNG$function type)
    (CNG$signature type infomorphism SET.FTN$function)
    (forall (?f (infomorphism ?f))
        (and (= (SET.FTN$source (type ?f)) (CLS$type (source ?f)))
             (= (SET.FTN$target (type ?f)) (CLS$type (target ?f)))))
```

- Axiom (6) gives the KIF for the fundamental property of an infomorphism.

```
(6) (forall (?f (infomorphism ?f)
             ?i ((CLS$instance (target ?f)) ?i)
             ?t ((CLS$type (source ?f)) ?t))
        (<=> ((CLS$incidence (source ?f)) [((instance ?f) ?i) ?t])
             ((CLS$incidence (target ?f)) [?i ((type ?f) ?t)])))
```

- The fundamental property of an infomorphism $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ expresses the bonding classification of a bond $\mathit{bond}(\boldsymbol{f}) : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ associated with the infomorphism. This can be equivalently define in terms of either the instance or the type function. Here we use the instance function. For comparison, see the right operator that maps a function to a relation in the presence of a preorder.

```
(7) (CNG$function bond)
    (CNG$signature bond infomorphism CLS.BND$bond)
    (forall (?f (infomorphism ?f))
        (and (= (CLS.BND$source (bond ?f)) (source ?f))
             (= (CLS.BND$target (bond ?f)) (target ?f))
```

```
            (forall (?i ((CLS$instance (target ?f)) ?i)
                        ?t ((CLS$type (source ?f)) ?t))
            (<=> ((CLS$incidence (CLS.BND$classification (bond ?f))) [?i ?t])
                 ((CLS$incidence (source ?f)) [((instance ?f) ?i) ?t]))))
```

- The instance function is a monotonic function between instance orders. Dually, the type function is a monotonic function between type orders. Axiom (8) defines the monotonic instance function in KIF, and axiom (9) does the same for the monotonic type function.

```
(8) (CNG$function monotonic-instance)
    (CNG$signature monotonic-instance infomorphism ORD.FTN$monotonic-function)
    (forall (?f (infomorphism ?f))
        (and (= (ORD.FTN$source (monotonic-instance ?f))
                (instance-order (target ?f)))
             (= (ORD.FTN$target (monotonic-instance ?f))
                (instance-order (source ?f)))
             (= (ORD.FTN$function (monotonic-instance ?f))
                (instance ?f))))

(9) (CNG$function monotonic-type)
    (CNG$signature monotonic-type infomorphism ORD.FTN$monotonic-function)
    (forall (?f (infomorphism ?f))
        (and (= (ORD.FTN$source (monotonic-type ?f))
                (type-order (source ?f)))
             (= (ORD.FTN$target (monotonic-type ?f))
                (type-order (target ?f)))
             (= (ORD.FTN$function (monotonic-type ?f))
                (type ?f))))
```

- The instance relation is the left relation of the monotonic instance function. The type relation is the right relation of the monotonic type function. Axiom (10) defines the instance relation of an infomorphism, and axiom (11) defines the type relation of an infomorphism.

```
(10) (CNG$function relational-instance)
     (CNG$signature relational-instance infomorphism REL$relation)
     (forall (?f (infomorphism ?f))
         (and (= (REL$source (relational-instance ?f))
                 (instance (source ?f)))
              (= (REL$target (relational-instance ?f))
                 (instance (target ?f)))
              (= (relational-instance ?f)
                 ((ORD.FTN$left (instance-order (source ?f)))
                     (monotonic-instance ?f))))))

(11) (CNG$function relational-type)
     (CNG$signature relational-type infomorphism REL$relation)
     (forall (?f (infomorphism ?f))
         (and (= (REL$source (relational-type ?f))
                 (type (source ?f)))
              (= (REL$target (relational-type ?f))
                 (type (target ?f)))
              (= (relational-type ?f)
                 ((ORD.FTN$right (type-order (target ?f)))
                     (monotonic-type ?f))))))
```

- Any functional infomorphism can be transformed into a relational infomorphism. In axiom (12) this *relational infomorphism* is defined below in terms of the instance and type relations.

```
(12) (CNG$function relational-infomorphism)
     (CNG$function fn2rel)
     (= fn2rel relational-infomorphism)
     (CNG$signature relational-infomorphism infomorphism CLS.REL$infomorphism)
     (forall (?f (infomorphism ?f))
         (and (= (CLS.REL$source (relational-infomorphism ?f))
                 (source ?f))
              (= (CLS.REL$target (relational-infomorphism ?f))
                 (target ?f))
              (= (CLS.REL$instance (relational-infomorphism ?f))
                 (relational-instance ?f))
              (= (CLS.REL$type (relational-infomorphism ?f))
```

```
                    (relational-type ?f))))
```

- It can be shown that the bond of the relational infomorphism of a functional infomorphism ***f*** is the bond associated with ***f***.

```
(forall (?f (infomorphism ?f))
    (= (CLS.REL$bond (relational-infomorphism ?f))
       (bond ?f)))
```

- The function ‘`composition`’ operates on any two infomorphisms that are composable in the sense that the target classification of the first is equal to the source classification of the second. Composition, defined in axiom (13), produces an infomorphism, whose instance function is the composition of the instance functions of the components and whose type function is the composition of the type functions of the components.

```
(13) (CNG$function composition)
     (CNG$signature composition infomorphism infomorphism infomorphism)
     (forall (?f (infomorphism ?f) ?g (infomorphism ?g))
         (<=> (exists (?h (infomorphism ?h)) (= (composition ?f ?g) ?h))
              (= (target ?f) (source ?g))))
     (forall (?f (infomorphism ?f) ?g (infomorphism ?g))
         (=> (= (target ?f) (source ?g))
             (and (= (source (composition ?f ?g)) (source ?f))
                  (= (target (composition ?f ?g)) (target ?g))
                  (= (instance (composition ?f ?g))
                     (SET.FTN$composition (instance ?g) (instance ?f)))
                  (= (type (composition ?f ?g))
                     (SET.FTN$composition (type ?f) (type ?g))))))
```

- The function ‘`identity`’ defined in axiom (14) associates a well-defined (identity) infomorphism with any classification, whose instance function is the identity class function on instances and whose type function is the identity class function on types.

```
(14) (CNG$function identity)
     (CNG$signature identity CLS$classification infomorphism)
     (forall (?c (CLS$classification ?c))
         (and (= (source (identity ?c)) ?c)
              (= (target (identity ?c)) ?c)
              (= (instance (identity ?c))
                 (SET.FTN$identity (CLS$instance ?c)))
              (= (type (identity ?c))
                 (SET.FTN$identity (CLS$type ?c)))))
```

- Duality can be extended to infomorphisms. For any infomorphism $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$, the *opposite* or *dual* of ***f*** is the infomorphism $\boldsymbol{f}^{\perp} : \boldsymbol{B}^{\perp} \rightleftarrows \boldsymbol{A}^{\perp}$ whose source classification is the opposite of the target classification of ***f***, whose target classification is the opposite of the source classification of ***f***, whose instance function is the type function of ***f***, whose type function is the instance function of ***f***, and whose fundamental condition is equivalent to that of ***f***:

  $\mathit{typ}(f)(t) \models^{\perp} i$ iff $t \models^{\perp} \mathit{inst}(f)(i)$.

  Axiom (15) specifies the opposite operator on infomorphisms.

```
(15) (CNG$function opposite)
     (CNG$signature opposite infomorphism infomorphism)
     (forall (?f (infomorphism ?f))
         (and (= (source (opposite ?f)) (CLS$opposite (target ?f)))
              (= (target (opposite ?f)) (CLS$opposite (source ?f)))
              (= (instance (opposite ?f)) (type ?f))
              (= (type (opposite ?f)) (instance ?f))))
```

- For any class function $f : B \to A$ the components of the *instance power infomorphism* $\wp f : \wp A \rightleftarrows \wp B$ over f are defined as follows: the source classification $\wp A = \langle A, \wp A, \in_A \rangle$ is the instance power classification over the target class A, the target classification $\wp B = \langle B, \wp B, \in_B \rangle$ is the instance power classification over the source class B, the instance function is f, and the type function is the inverse image function $f^{-1} : \wp A \to \wp B$ from the power-class of A to the power-class of B. Note the contravariance.

```
(16) (CNG$function instance-power)
     (CNG$signature instance-power SET.FTN$function infomorphism)
     (forall (?f (SET.FTN$function ?f))
         (and (= (source (instance-power ?f)) (SET.FTN$power (SET.FTN$target ?f)))
              (= (target (instance-power ?f)) (SET.FTN$power (SET.FTN$source ?f)))
              (= (instance (instance-power ?f)) ?f)
              (= (type (instance-power ?f))
                 (SET.FTN$inverse-image ?f))))
```

- It is a standard fact in Information Flow that from any classification $\boldsymbol{A}$ there is a *canonical extent infomorphism* $\eta_{\boldsymbol{A}} : \boldsymbol{A} \rightleftarrows \wp inst(\boldsymbol{A})$ from $\boldsymbol{A}$ to the instance power classification $\wp inst(\boldsymbol{A}) = \langle inst(\boldsymbol{A}), \wp inst(\boldsymbol{A}), \in \rangle$ over the instance class. This infomorphism is the $\boldsymbol{A}^{\text{th}}$ component of a natural quasi-transformation called *eta*. The instance function is the identity function on the instance class $inst(\boldsymbol{A})$, and the type function is the extent function $extent_{\boldsymbol{A}} : typ(\boldsymbol{A}) \to \wp inst(\boldsymbol{A})$.

  Axiom (17) gives the KIF definition of the extent infomorphism.

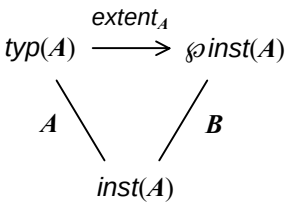


Figure 2: $\eta_{\boldsymbol{A}}$, the extent infomorphism

```
(17) (CNG$function eta)
     (CNG$signature eta CLS$classification infomorphism)
     (forall (?a (classification ?a))
         (and (= (source (eta ?a)) ?a)
              (= (target (eta ?a))
                 (CLS$instance-power (CLS$instance ?a)))
              (= (instance (eta ?a))
                 (SET.FTN$identity (CLS$instance ?a)))
              (= (type (eta ?a)) (CLS$extent ?a))))
```

- Let $\boldsymbol{f} = \langle inst(\boldsymbol{f}), typ(\boldsymbol{f}) \rangle : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ be any infomorphism from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ with instance function $inst(\boldsymbol{f}) : inst(\boldsymbol{B}) \to inst(\boldsymbol{A})$ and type function $typ(\boldsymbol{f}) : typ(\boldsymbol{A}) \to typ(\boldsymbol{B})$. There is an adjoint pair

  $$adj(\boldsymbol{f}) = \langle left(adj(\boldsymbol{f})), right(adj(\boldsymbol{f})) \rangle : complete\text{-}lattice(\boldsymbol{A}) \rightleftarrows complete\text{-}lattice(\boldsymbol{B}),$$

  defined as follows.

  $$\begin{aligned} &left(adj(\boldsymbol{f}))(d) \\ &\quad = left(adj(\boldsymbol{f}))(\langle extent_{\boldsymbol{B}}(d), intent_{\boldsymbol{B}}(d) \rangle) \\ &\quad = \langle (typ(\boldsymbol{f})^{-1}(intent_{\boldsymbol{B}}(d)))', \ (typ(\boldsymbol{f})^{-1}(intent_{\boldsymbol{B}}(d))) \rangle \end{aligned}$$

  for all concepts $d \in complete\text{-}lattice(\boldsymbol{B})$, and

  $$\begin{aligned} &right(adj(\boldsymbol{f}))(c) \\ &\quad = right(adj(\boldsymbol{f}))(\langle extent_{\boldsymbol{A}}(c), intent_{\boldsymbol{A}}(c) \rangle) \\ &\quad = \langle (inst(\boldsymbol{f})^{-1}(extent_{\boldsymbol{A}}(c)), (inst(\boldsymbol{f})^{-1}(extent_{\boldsymbol{A}}(c)))' \rangle \end{aligned}$$

  for all concepts $c \in complete\text{-}lattice(\boldsymbol{A})$.

```
(18) (CNG$function adjoint-pair)
     (CNG$signature adjoint-pair infomorphism LAT.ADJ$adjoint-pair)
     (forall (?f (infomorphism ?f))
         (and (= (LAT.ADJ$source (adjoint-pair ?f))
                 (complete-lattice (source ?f)))
              (= (LAT.ADJ$target (adjoint-pair ?f))
                 (complete-lattice (target ?f)))
              (= (SET.FTN$composition
                     (ORD.FTN$function (LAT.ADJ$left (adjoint-pair ?f)))
                     (CLS.CL$intent (source ?f)))
                 (SET.FTN$composition
                     (CLS.CL$intent (target ?f))
                     (SET.FTN$inverse-image (type ?f))))
              (= (SET.FTN$composition
                     (ORD.FTN$function (LAT.ADJ$right (adjoint-pair ?f)))
                     (CLS.CL$extent (target ?f)))
                 (SET.FTN$composition
```

```
                    (CLS.CL$extent (source ?f))
                    (SET.FTN$inverse-image (instance ?f))))))
```

- Let $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ be any infomorphism from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ with instance function $\mathit{inst}(\boldsymbol{f}) : \mathit{inst}(\boldsymbol{B}) \to \mathit{inst}(\boldsymbol{A})$ and type function $\mathit{typ}(\boldsymbol{f}) : \mathit{typ}(\boldsymbol{A}) \to \mathit{typ}(\boldsymbol{B})$. There is a concept morphism

  $$\mathit{concept\text{-}morphism}(\boldsymbol{f}) = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \mathit{adj}(\boldsymbol{A}) \rangle : \mathit{concept\text{-}lattice}(\boldsymbol{A}) \rightleftarrows \mathit{concept\text{-}lattice}(\boldsymbol{B}),$$

  whose instance/type functions are the same as $\boldsymbol{f}$, and whose adjoint pair the adjoint pair of $\boldsymbol{f}$.

```
(19) (CNG$function concept-morphism)
     (CNG$signature concept-morphism CLS.INFO$infomorphism CL.MOR$concept-morphism)
     (forall (?f (CLS.INFO$infomorphism ?f))
         (and (= (CL.MOR$source (concept-morphism ?f))
                 (concept-lattice (CLS.INFO$source ?f)))
              (= (CL.MOR$target (concept-morphism ?f))
                 (concept-lattice (CLS.INFO$target ?f)))
              (= (CL.MOR$adjoint-pair (concept-morphism ?f)) (adjoint-pair ?f))
              (= (CL.MOR$instance (concept-morphism ?f)) (instance ?f))
              (= (CL.MOR$type (concept-morphism ?f)) (type ?f))))
```

Relational Infomorphisms

CLS.REL

- Classifications are also related through (relational) infomorphisms. A (*relational*) *infomorphism* $\boldsymbol{r} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$ (see Figure 5) is a pair $\boldsymbol{r} = \langle inst(\boldsymbol{r}), typ(\boldsymbol{r}) \rangle$ of binary relations, a relation between instances $inst(\boldsymbol{r}) : inst(\boldsymbol{A}) \to inst(\boldsymbol{B})$ and a relation between types $typ(\boldsymbol{r}) : typ(\boldsymbol{A}) \to typ(\boldsymbol{B})$, which satisfy the fundamental property:

  $inst(\boldsymbol{r}) \setminus \boldsymbol{A} = \boldsymbol{B} / typ(\boldsymbol{r})$.

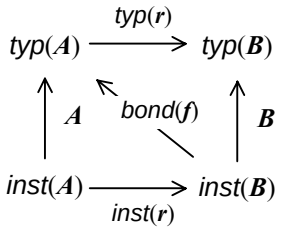


Figure 5: Relational Infomorphism

The following is a KIF representation for the elements of a relation infomorphism.

```
(1) (CNG$conglomeration infomorphism)

(2) (CNG$function source)
    (CNG$signature source infomorphism CLS$classification)

(3) (CNG$function target)
    (CNG$signature target infomorphism CLS$classification)

(4) (CNG$function instance)
    (CNG$signature instance infomorphism REL$relation)
    (forall (?r (infomorphism ?r))
        (and (= (REL$source (instance ?r)) (CLS$instance (source ?r)))
             (= (REL$target (instance ?r)) (CLS$instance (target ?r)))))

(5) (CNG$function type)
    (CNG$signature type infomorphism REL$relation)
    (forall (?r (infomorphism ?r))
        (and (= (REL$source (type ?r)) (CLS$type (source ?r)))
             (= (REL$target (type ?r)) (CLS$type (target ?r)))))
```

- Axiom (6) gives the KIF for the fundamental property of a relational infomorphism.

```
(6) (forall (?r (infomorphism ?r))
        (= (REL$left-residuation (instance ?r) (CLS$incidence (source ?r)))
           (REL$right-residuation (type ?r) (CLS$incidence (target ?r)))))
```

- For any relational infomorphism $\boldsymbol{r} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$, the common relation $bond(\boldsymbol{r}) = inst(\boldsymbol{r}) \backslash \boldsymbol{A} = \boldsymbol{B} / typ(\boldsymbol{r}) : inst(\boldsymbol{A}) \to inst(\boldsymbol{B})$ in the fundamental property is called the *bond* of $\boldsymbol{r}$ – it bonds the instance and type relations into a unity.

```
(7) (CNG$function bond)
    (CNG$signature bond infomorphism CLS.BND$bond)
    (forall (?r (infomorphism ?r))
        (and (= (CLS.BND$source (bond ?r) (source ?r))
             (= (CLS.BND$target (bond ?r) (target ?r))
             (= (CLS.BND$classification (bond ?r)
                (REL$left-residuation (instance ?r) (CLS$incidence (source ?r))))))
```

- Given any two relational infomorphisms $\langle r_1, s_1 \rangle : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ and $\langle r_2, s_2 \rangle : \boldsymbol{B} \rightleftarrows \boldsymbol{C}$, which are composable in the sense the target classification of the first is the source classification of the second, there is a *composite infomorphism* $\langle r_1, s_1 \rangle \circ \langle r_2, s_2 \rangle = \langle r_1 \circ r_2, s_1 \circ s_2 \rangle : \boldsymbol{A} \rightleftarrows \boldsymbol{C}$ defined by composing the type and instance relations, and whose fundamental property follows from composition and associative laws.

```
(8) (CNG$function composition)
    (CNG$signature composition infomorphism infomorphism infomorphism)
    (forall (?r (infomorphism ?r) ?s (infomorphism ?s))
        (<=> (exists (?t (infomorphism ?t)) (= (composition ?r ?s) ?t))
             (= (target ?r) (source ?s))))
    (forall (?r (infomorphism ?r) ?s (infomorphism ?s))
        (=> (= (target ?r) (source ?s))
            (and (= (source (composition ?r ?s)) (source ?r))
```

```
(= (target (composition ?r ?s)) (target ?s))
(= (instance (composition ?r ?s))
   (REL$composition (instance ?r) (instance ?s)))
(= (type (composition ?r ?s))
   (REL$composition (type ?r) (type ?s))))))
```

- Given any classification $\boldsymbol{A} = \langle inst(\boldsymbol{A}), typ(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, the pair of identity relations on types and instances, with the bond being $\boldsymbol{A}$, forms an *identity infomorphism* $Id_{\boldsymbol{A}} : \boldsymbol{A} \rightleftarrows \boldsymbol{A}$ (with respect to composition).

```
(9) (CNG$function identity)
    (CNG$signature identity CLS$classification infomorphism)
    (forall (?a (CLS$classification ?a))
        (and (= (source (identity ?a)) ?a)
             (= (target (identity ?a)) ?a)
             (= (instance (identity ?a)) (REL$identity (CLS$instance ?a)))
             (= (type (identity ?a)) (REL$identity (CLS$type ?a)))))
```

- For any given infomorphism $\langle r, s \rangle : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ the *dual infomorphism* $\langle r, s \rangle^{op} = \langle s^{op}, r^{op} \rangle : \boldsymbol{B}^{op} \rightleftarrows \boldsymbol{A}^{op}$ is the relational infomorphism with type and instance relations switched and transposed.

```
(10) (CNG$function opposite)
     (CNG$signature opposite infomorphism infomorphism)
     (forall (?r (infomorphism ?r))
         (and (= (source (opposite ?r)) (CLS$opposite (target ?r)))
              (= (target (opposite ?r)) (CLS$opposite (source ?r)))
              (= (instance (opposite ?r)) (REL$opposite (type ?r)))
              (= (type (opposite ?r)) (REL$opposite (instance ?r)))))
```

- The fundamental property of relational infomorphisms for composition, identity and involution follow from basic properties of residuation.

Bonds

CLS.BND

- Classifications are also related through bonds. A *bond* $\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{B}$ from classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$ to classification $\boldsymbol{B} = \langle \mathit{inst}(\boldsymbol{B}), \mathit{typ}(\boldsymbol{B}), \models_{\boldsymbol{B}} \rangle$ (see Figure 6) is a classification $\mathit{cls}(\boldsymbol{F}) = \langle \mathit{inst}(\boldsymbol{B}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{F}} \rangle$ sharing types with $\boldsymbol{A}$ and instances with $\boldsymbol{B}$, that is compatible with $\boldsymbol{A}$ and $\boldsymbol{B}$ in the sense of closure: type sets $\{j\boldsymbol{F} \mid j \in \mathit{inst}(\boldsymbol{B})\}$ are intents of $\boldsymbol{A}$ and instance sets $\{\boldsymbol{F}t \mid t \in \mathit{typ}(\boldsymbol{B})\}$ are extents of $\boldsymbol{B}$. Closure can be expressed relationally (in terms of residuation) as the following fundamental properties.

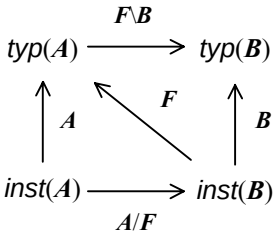


Figure 6: Bond

  $(\boldsymbol{A}/\boldsymbol{F})\backslash\boldsymbol{A} = \boldsymbol{F}$ and $\boldsymbol{B}/(\boldsymbol{F}\backslash\boldsymbol{B}) = \boldsymbol{F}$.

  The first expression says that $\langle \boldsymbol{A}/\boldsymbol{F}, \boldsymbol{F} \rangle$ is an $\mathit{inst}(\boldsymbol{B})$-indexed collective $\boldsymbol{A}$-concept (or $\boldsymbol{F}$ is a collective $\boldsymbol{A}$-intent), and the second expression says that $\langle \boldsymbol{F}, \boldsymbol{F}\backslash\boldsymbol{B} \rangle$ is an $\mathit{typ}(\boldsymbol{A})$-indexed collective $\boldsymbol{B}$-concept (or that $\boldsymbol{F}$ is a collective $\boldsymbol{B}$-extent).

```
(1) (CNG$conglomeration bond)

(2) (CNG$function source)
    (CNG$signature source bond CLS$classification)

(3) (CNG$function target)
    (CNG$signature target bond CLS$classification)

(4) (CNG$function classification)
    (CNG$signature classification bond CLS$classification)
    (forall (?b (bond ?b))
        (and (= (CLS$instance (classification ?b)) (CLS$instance (target ?b)))
             (= (CLS$type (classification ?b)) (CLS$type (source ?b)))))
```

- Axiom (5) gives the KIF for the closure properties required of a bond.

```
(5) (forall (?b (bond ?b))
        (and (= (REL$left-residuation
                    (REL$right-residuation (classification ?b) (source ?b))
                    (source ?b))
                (classification ?b))
             (= (REL$right-residuation
                    (REL$left-residuation (classification ?b) (target ?b))
                    (target ?b))
                (classification ?b))))
```

- (The classification relation of) a bond is order-closed on left and right:

  $j' \leq_{\boldsymbol{B}} j, jFt$ imply $j'Ft$, and $jFt, t \leq_{\boldsymbol{A}} t'$ imply jFt'. Or,

  $j' \leq_{\boldsymbol{B}} j$ implies $j'F \supseteq jF$, and $t \leq_{\boldsymbol{A}} t'$ implies $Ft \subseteq Ft'$.

  The '`bimodule`' function defined in axiom (6) realizes these properties.

```
(6) (CNG$function bimodule)
    (CNG$signature bimodule bond ORD$bimodule)
    (forall (?b (bond ?b))
        (and (= (ORD$source (bimodule ?b))
                (CLS.CL$instance-order (target ?b)))
             (= (ORD$target (bimodule ?b))
                (CLS.CL$type-order (source ?b)))
             (= (ORD$relation (bimodule ?b))
                (classification ?b)))))
```

- For any bond $\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{B}$, the residuations in the fundamental closure property form a relation infomorphism $\mathit{info}(\boldsymbol{F}) = \langle (\boldsymbol{A}/\boldsymbol{F}), (\boldsymbol{F}\backslash\boldsymbol{B}) \rangle : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$.

```
(7) (CNG$function infomorphism)
    (CNG$signature infomorphism bond CLS.REL$infomorphism)
```

```
(forall (?b (bond ?b))
    (and (= (CLS.REL$source (infomorphism ?b)) (source ?b))
         (= (CLS.REL$target (infomorphism ?b)) (target ?b))
         (= (CLS.REL$instance (infomorphism ?b))
            (REL$right-residuation (classification ?b) (source ?b)))
         (= (CLS.REL$type (infomorphism ?b))
            (REL$left-residuation (classification ?b) (target ?b)))))
```

- Moreover, the fundamental closure property implies that the bond of this relational infomorphism is the original bond.

```
(forall (?b (bond ?b))
    (= (CLS.REL$bond (infomorphism ?b)) ?b))
```

- Associated with any bond is a morphism (*adjoint pair*) of the complete lattices of its source and target classifications.

  – By the first fundamental property, the two classifications $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$ and $\boldsymbol{F} = \langle \mathit{inst}(\boldsymbol{B}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{F}} \rangle$ have the same class of types, and the classification (binary relation) $\boldsymbol{F}$ is *type-closed* $(\boldsymbol{A}/\boldsymbol{F})\backslash\boldsymbol{A} = \boldsymbol{F}$ with respect to $\boldsymbol{A}$. Hence, there is an associated *coreflection* (adjoint pair) $\mathit{corefl}_{\boldsymbol{A},\boldsymbol{F}} = \langle \partial_0, \tilde{\partial}_0 \rangle : \mathit{latt}(\boldsymbol{F}) \rightleftarrows \mathit{latt}(\boldsymbol{A})$ between their complete lattices, where $\tilde{\partial}_0 : \mathit{latt}(\boldsymbol{A}) \to \mathit{latt}(\boldsymbol{F})$ is right adjoint right inverse (rari) to $\partial_0 : \mathit{latt}(\boldsymbol{F}) \to \mathit{latt}(\boldsymbol{A})$.

  – By the second fundamental property, the two classifications $\boldsymbol{F} = \langle \mathit{inst}(\boldsymbol{B}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{F}} \rangle$ and $\boldsymbol{B} = \langle \mathit{inst}(\boldsymbol{B}), \mathit{typ}(\boldsymbol{B}), \models_{\boldsymbol{B}} \rangle$ have the same class of instances, and the classification (binary relation) $\boldsymbol{F}$ is *instance-closed* $\boldsymbol{B}/(\boldsymbol{F}\backslash\boldsymbol{B}) = \boldsymbol{F}$ with respect to $\boldsymbol{B}$. Hence, there is an associated *reflection* (adjoint pair) $\mathit{refl}_{\boldsymbol{F},\boldsymbol{B}} = \langle \tilde{\partial}_1, \partial_1 \rangle : \mathit{latt}(\boldsymbol{B}) \rightleftarrows \mathit{latt}(\boldsymbol{F})$ between their complete lattices, where $\tilde{\partial}_1 : \mathit{latt}(\boldsymbol{B}) \to \mathit{latt}(\boldsymbol{F})$ is left adjoint right inverse (lari) to $\partial_1 : \mathit{latt}(\boldsymbol{F}) \to \mathit{latt}(\boldsymbol{B})$.

  Define the associated adjoint pair as the composition of the reflection followed by the coreflection.

```
(8) (CNG$function adjoint-pair)
    (CNG$signature adjoint-pair bond LAT.ADJ$adjoint-pair)
    (forall (?b (bond ?b))
        (and (= (LAT.ADJ$source (adjoint-pair ?b))
                (CLS.CL$complete-lattice (target ?b)))
             (= (LAT.ADJ$target (adjoint-pair ?b))
                (CLS.CL$complete-lattice (source ?b)))
             (= (LAT.ADJ$underlying (adjoint-pair ?b))
                (LAT.ADJ$composition
                    (CLS.CL$reflection (target ?b) (classification ?b))
                    (CLS.CL$coreflection (classification ?b) (source ?b)))))))
```

- Two bonds are composable when the target classification of the first is the source classification of the second. For any two composable bonds $\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{B}$ and $\boldsymbol{G} : \boldsymbol{B} \multimap \boldsymbol{C}$, the *composition* is the bond $\boldsymbol{F} \circ \boldsymbol{G} \triangleq (\boldsymbol{B}/\boldsymbol{G})\backslash\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{C}$ defined using left and right residuation. Since both $\boldsymbol{F}$ and $\boldsymbol{G}$ being bonds are closed with respect to $\boldsymbol{B}$, an equivalent expression for the composition is $\boldsymbol{F} \circ \boldsymbol{G} \triangleq \boldsymbol{G}/(\boldsymbol{F}\backslash\boldsymbol{B}) : \boldsymbol{A} \multimap \boldsymbol{C}$. Pointwise, the composition is $\boldsymbol{F} \circ \boldsymbol{G} = \{(c, \alpha) \mid \boldsymbol{F}\alpha \supseteq (c\boldsymbol{G})^{\boldsymbol{B}}\}$. To check closure, $(\boldsymbol{A}/(\boldsymbol{F} \circ \boldsymbol{G}))\backslash\boldsymbol{A} = (\boldsymbol{A}/((\boldsymbol{B}/\boldsymbol{G})\backslash\boldsymbol{F}))\backslash\boldsymbol{A} = (\boldsymbol{B}/\boldsymbol{G})\backslash\boldsymbol{F}$, since $\boldsymbol{F}$ being a collective $\boldsymbol{A}$-intent means that $(\boldsymbol{B}/\boldsymbol{G})\backslash\boldsymbol{F}$ is also a collective $\boldsymbol{A}$-intent.

- Composition, defined in axiom (8), produces a bond constructed according to the expressions above.

```
(8) (CNG$function composition)
    (CNG$signature composition bond bond bond)
    (forall (?f (bond ?f) ?g (bond ?g))
        (<=> (exists (?h (bond ?h)) (= (composition ?f ?g) ?h))
             (= (target ?f) (source ?g))))
    (forall (?f (bond ?f) ?g (bond ?g))
        (=> (= (target ?f) (source ?g))
            (and (= (source (composition ?f ?g)) (source ?f))
                 (= (target (composition ?f ?g)) (target ?g))
                 (= (classification (composition ?f ?g))
                    (REL$left-residuation (REL$right-residuation ?g (source ?g)) ?f)))))
```

- With respect to bond composition, the *identity* bond at any classification $\boldsymbol{A}$ is $\boldsymbol{A}$.

```
(9) (CNG$function identity)
    (CNG$signature identity CLS$classification bond)
    (forall (?a (CLS$classification ?a))
        (and (= (source (identity ?a)) ?a)
             (= (target (identity ?a)) ?a)
             (= (classification (identity ?a)) ?a)))
```

- For any given bond $\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{B}$ the *dual bond* $\boldsymbol{F}^{\mathrm{op}} : \boldsymbol{B}^{\mathrm{op}} \multimap \boldsymbol{A}^{\mathrm{op}}$ is the bond with source and target classifications switched, and source, target and classification relations transposed.

```
(10) (CNG$function opposite)
     (CNG$signature opposite bond bond)
     (forall (?f (bond ?f))
         (and (= (source (opposite ?f)) (CLS$opposite (target ?f)))
              (= (target (opposite ?f)) (CLS$opposite (source ?f)))
              (= (classification (opposite ?f)) (REL$opposite (classification ?f)))))
```

The fundamental property of bonds for composition, identity and involution follow from basic properties of residuation.

- The basic theorem of Formal Concept Analysis can be framed in terms of two fundamental bonds (relational infomorphisms) between any classification and its associated concept lattice.
  - For any classification $\boldsymbol{A}$ the instance embedding relation is a bond $\iota_{\boldsymbol{A}} : \boldsymbol{L}(\boldsymbol{A}) \multimap \boldsymbol{A}$ named *iota*, whose source classification is the (classification of the) concept lattice $\boldsymbol{L}(\boldsymbol{A})$ and whose target classification is $\boldsymbol{A}$. The classification relation of the instance embedding bond is the instance embedding relation. The pair $\langle \boldsymbol{L}(\boldsymbol{A})/\iota_{\boldsymbol{A}}, \tau_{\boldsymbol{A}} \rangle : \boldsymbol{L}(\boldsymbol{A}) \rightleftharpoons \boldsymbol{A}$ is a relational infomorphism whose bond is the iota bond.
  - For any classification $\boldsymbol{A}$ the type embedding relation is a bond $\tau_{\boldsymbol{A}} : \boldsymbol{A} \multimap \boldsymbol{L}(\boldsymbol{A})$ named *tau*, whose source classification is $\boldsymbol{A}$ and whose target classification is the (classification of the) concept lattice $\boldsymbol{L}(\boldsymbol{A})$. The classification relation of the type embedding bond is the type embedding relation. The pair $\langle \iota_{\boldsymbol{A}}, \tau_{\boldsymbol{A}} \backslash \boldsymbol{L}(\boldsymbol{A}) \rangle : \boldsymbol{A} \rightleftharpoons \boldsymbol{L}(\boldsymbol{A})$ is a relational infomorphism whose bond is the type embedding relation.

```
(11) (CNG$function iota)
     (CNG$signature iota CLS$classification bond)
     (forall (?a (CLS$classification ?a))
         (and (= (source (iota ?a)) (CL$classification (CLS.CL$concept-lattice ?a)))
              (= (type (iota ?a)) ?a)
              (= (classification (iota ?a)) CLS.CL$iota)))

(12) (CNG$function tau)
     (CNG$signature tau CLS$classification bond)
     (forall (?a (CLS$classification ?a))
         (and (= (source (iota ?a)) ?a)
              (= (type (iota ?a)) (CL$classification (CLS.CL$concept-lattice ?a)))
              (= (classification (tau ?a)) CLS.CL$tau)))
```

- The instance and type embedding bonds are inverse to each other: $\iota_{\boldsymbol{A}} \circ \tau_{\boldsymbol{A}} = Id_{L(\boldsymbol{A})}$ and $\tau_{\boldsymbol{A}} \circ \iota_{\boldsymbol{A}} = Id_{\boldsymbol{A}}$.

```
(forall (?a (CLS$classification ?a))
    (and (= (composition (iota ?a) (tau ?a))
            (identity (CL$classification (CLS.CL$concept-lattice ?a))))
         (= (composition (tau ?a) (iota ?a))
            (identity ?a))))
```

- The functor composition $\boldsymbol{A} \circ \boldsymbol{B}$ is naturally isomorphic to the identity functor $Id_{\mathbf{Bond}}$. To see this, let $\boldsymbol{A}$ be a classification with associated concept lattice $\mathit{lat}(\boldsymbol{A})$. The comments above demonstrate the bond isomorphism $\boldsymbol{A} \cong \mathit{cls}(\mathit{lat}(\boldsymbol{A}))$. Let $\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{B}$ be a bond between classifications $\boldsymbol{A}$ and $\boldsymbol{B}$ with associated complete adjoint

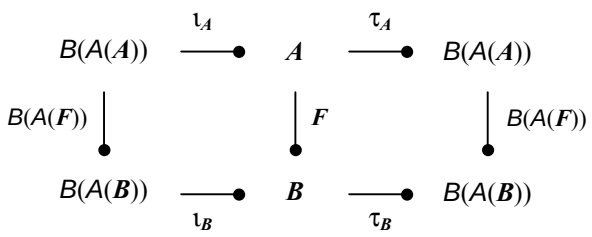


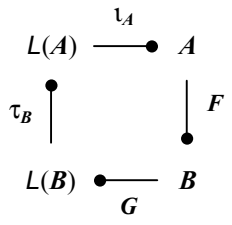
Diagram 2: Natural Isomorphism

$adj(\boldsymbol{F}) : lat(\boldsymbol{A}) \rightleftarrows lat(\boldsymbol{B})$. The classification relation of the bond $bnd(adj(\boldsymbol{F})) : cls(lat(\boldsymbol{A})) \multimap cls(lat(\boldsymbol{B}))$ contains a conceptual pair $(\boldsymbol{a}, \boldsymbol{b})$ of the form $\boldsymbol{a} = (A, \Gamma)$ and $\boldsymbol{b} = (B, \Delta)$ iff $B \circ \Gamma \subseteq \boldsymbol{F}$, where $B \circ \Gamma = B \times \Gamma$ a Cartesian product or rectangle, iff $B \subseteq \Gamma^{\boldsymbol{F}}$ iff $\Gamma \subseteq B^{\boldsymbol{F}}$. So, $(\iota_{\boldsymbol{A}} \boxdot \boldsymbol{F}) \boxdot \tau_{\boldsymbol{B}} = (\boldsymbol{F}/(\iota_{\boldsymbol{A}} \backslash \boldsymbol{A})) \boxdot \tau_{\boldsymbol{B}} = (\boldsymbol{F}/\tau_{\boldsymbol{A}}) \boxdot \tau_{\boldsymbol{B}} = (\boldsymbol{B}/\tau_{\boldsymbol{B}}) \backslash (\boldsymbol{F}/\tau_{\boldsymbol{A}}) = \iota_{\boldsymbol{B}} \backslash (\boldsymbol{F}/\tau_{\boldsymbol{A}}) = bnd(adj(\boldsymbol{F}))$, by bond composition and properties of the instance and type relations. Hence, $bnd(adj(\boldsymbol{F})) \boxdot \iota_{\boldsymbol{B}} = \iota_{\boldsymbol{A}} \boxdot \boldsymbol{F}$. This proves the required naturality condition.

```
(forall (?f (bond ?f))
    (= (composition (CL.MOR$bond (adjoint-pair ?f)) (iota (target ?f)))
       (composition (iota (source ?f)) ?f)))
```

Bonding Pairs

CLS.BNDPR



A *complete* (*lattice*) *homomorphism* $\psi : \boldsymbol{L} \to \boldsymbol{K}$ between complete lattices $\boldsymbol{L}$ and $\boldsymbol{K}$ is a (monotonic) function that preserves both joins and meets. Being meet-preserving, ψ has a left adjoint $\varphi : \boldsymbol{K} \to \boldsymbol{L}$, and being join-preserving ψ has a right adjoint $\theta : \boldsymbol{K} \to \boldsymbol{L}$. Therefore, a complete homomorphism is the middle monotonic function in two adjunctions $\varphi \dashv \psi \dashv \theta$. Let Complete Lattice denote the quasi-category of complete lattices and complete homomorphisms.

The bond equivalent to a complete homomorphism would seem to be given by two bonds $\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{B}$ and $\boldsymbol{G} : \boldsymbol{B} \multimap \boldsymbol{A}$ where the right adjoint $\psi_{\boldsymbol{F}} : \boldsymbol{L}(\boldsymbol{A}) \to \boldsymbol{L}(\boldsymbol{B})$ of the complete adjoint $\boldsymbol{A}(\boldsymbol{F}) = \langle \varphi_{\boldsymbol{F}}, \psi_{\boldsymbol{F}} \rangle : \boldsymbol{L}(\boldsymbol{B}) \rightleftarrows \boldsymbol{L}(\boldsymbol{A})$ of one bond (say $\boldsymbol{F}$, without loss of generality) is equal to the left adjoint $\varphi_{\boldsymbol{G}} : \boldsymbol{L}(\boldsymbol{A}) \to \boldsymbol{L}(\boldsymbol{B})$ of the complete adjoint $\boldsymbol{A}(\boldsymbol{G}) = \langle \varphi_{\boldsymbol{G}}, \psi_{\boldsymbol{G}} \rangle : \boldsymbol{L}(\boldsymbol{A}) \rightleftarrows \boldsymbol{L}(\boldsymbol{B})$ of the other bond $\boldsymbol{G}$ with the resultant adjunctions, $\varphi_{\boldsymbol{F}} \dashv \psi_{\boldsymbol{F}} = \varphi_{\boldsymbol{G}} \dashv \psi_{\boldsymbol{G}}$, where the middle adjoint is the complete homomorphism. This is indeed the case, but the question is, what constraint should be placed on $\boldsymbol{F}$ and $\boldsymbol{G}$ in order for this to hold. The simple answer is to identify the actions of the two monotonic functions $\psi_{\boldsymbol{G}}$ and $\varphi_{\boldsymbol{F}}$. Let $(A, \Gamma) \in \boldsymbol{L}(\boldsymbol{B})$ be any formal concept in $\boldsymbol{L}(\boldsymbol{A})$. The action of the left adjoint $\varphi_{\boldsymbol{G}}$ on this concept is $(A, \Gamma) \mapsto (A^{\boldsymbol{GB}}, A^{\boldsymbol{G}})$, whereas the action of the right adjoint $\psi_{\boldsymbol{F}}$ on this concept is $(A, \Gamma) \mapsto (\Gamma^{\boldsymbol{F}}, \Gamma^{\boldsymbol{FB}})$. So the appropriate pointwise constraints are: $A^{\boldsymbol{GB}} = \Gamma^{\boldsymbol{F}}$ and $\Gamma^{\boldsymbol{FB}} = A^{\boldsymbol{G}}$, for every concept $(A, \Gamma) \in \boldsymbol{L}(\boldsymbol{A})$. We now give these pointwise constraints a relational formulation.

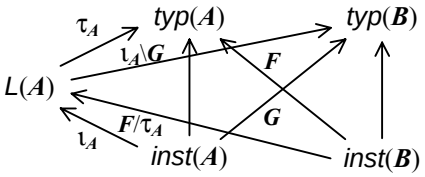


Figure 7: Bonding Pair

- A *bonding pair* $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \multimap \boldsymbol{B}$ between two classifications $\boldsymbol{A}$ and $\boldsymbol{B}$ is a contravariant pair of bonds, a bond $\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{B}$ in the *forward* direction and a bond $\boldsymbol{G} : \boldsymbol{B} \multimap \boldsymbol{A}$ in the *reverse* direction, satisfying the following *pairing constraints*:

  $\boldsymbol{F}/\tau_{\boldsymbol{A}} = \boldsymbol{B}/(\iota_{\boldsymbol{A}} \backslash \boldsymbol{G})$ and $\iota_{\boldsymbol{A}} \backslash \boldsymbol{G} = (\boldsymbol{F}/\tau_{\boldsymbol{A}}) \backslash \boldsymbol{B}$,

  which state that $(\boldsymbol{F}/\tau_{\boldsymbol{A}}, \iota_{\boldsymbol{A}} \backslash \boldsymbol{G}) = (\iota_{\boldsymbol{A}} \triangleright \boldsymbol{F}, \boldsymbol{G} \triangleright \tau_{\boldsymbol{B}})$ is an $\boldsymbol{L}(\boldsymbol{A})$-indexed collective $\boldsymbol{B}$-concept. The definitions of the relations $\boldsymbol{F}/\tau_{\boldsymbol{A}}$ and $\iota_{\boldsymbol{A}} \backslash \boldsymbol{G}$ are given as follows: $\boldsymbol{F}/\tau_{\boldsymbol{A}} = \{(b, \boldsymbol{a}) \mid int(\boldsymbol{a}) \subseteq b\boldsymbol{F}\} = \{(b, \boldsymbol{a}) \mid ((b\boldsymbol{F})^{\boldsymbol{A}}, b\boldsymbol{F}) \leq_{\boldsymbol{B}} \boldsymbol{a}\}$ and $\iota_{\boldsymbol{A}} \backslash \boldsymbol{G} = \{(\boldsymbol{a}, \beta) \mid ext(\boldsymbol{a}) \subseteq \boldsymbol{G}\beta\} = \{(\boldsymbol{a}, \beta) \mid \boldsymbol{a} \leq_{\boldsymbol{B}} (\boldsymbol{G}\beta, (\boldsymbol{G}\beta)^{\boldsymbol{A}})\}$. Any concept $\boldsymbol{a} = (A, \Gamma) \in \boldsymbol{L}(\boldsymbol{A})$ is mapped by the relations as: $(\boldsymbol{F}/\tau_{\boldsymbol{A}})((A, \Gamma)) = \{b \mid \Gamma \subseteq b\boldsymbol{F}\} = \Gamma^{\boldsymbol{F}}$ and $(\iota_{\boldsymbol{A}} \backslash \boldsymbol{G})((A, \Gamma)) = \{\beta \mid A \subseteq \boldsymbol{G}\beta\} = A^{\boldsymbol{G}}$. Hence, pointwise the constraints are $\Gamma^{\boldsymbol{F}} = B^{\boldsymbol{GB}}$ and $A^{\boldsymbol{G}} = \Gamma^{\boldsymbol{FB}}$. These are the pointwise constraints discussed above.

```
(1) (CNG$conglomeration bonding-pair)

(2) (CNG$function source)
    (CNG$signature source bonding-pair CLS$classification)

(3) (CNG$function target)
    (CNG$signature target bonding-pair CLS$classification)

(4) (CNG$function forward)
    (CNG$signature forward bonding-pair bond)
    (forall (?bp (bonding-pair ?bp))
        (and (= (CLS.BND$source (forward ?bp)) (source ?bp))
             (= (CLS.BND$target (forward ?bp)) (target ?bp))))

(5) (CNG$function reverse)
    (CNG$signature reverse bonding-pair bond)
    (forall (?bp (bonding-pair ?bp))
        (and (= (CLS.BND$source (reverse ?bp)) (target ?bp))
             (= (CLS.BND$target (reverse ?bp)) (source ?bp))))

    (forall (?bp (bonding-pair ?bp))
```

```
        (and (= (REL$right-residuation (CLS.CL$tau (source ?bp)) (forward ?bp))
                (REL$right-residuation
                    (REL$left-residuation (CLS.CL$iota (source ?bp)) (reverse ?bp))
                    (target ?bp)))
             (= (REL$left-residuation (CLS.CL$iota (source ?bp)) (reverse ?bp))
                (REL$left-residuation
                    (REL$right-residuation (CLS.CL$tau (source ?bp)) (forward ?bp))
                    (target ?bp)))
```

- The pointwise constraints can be lifted to a collective setting – any bonding pair $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \multimap \boldsymbol{B}$ preserves collective concepts: for any A-indexed collective $\boldsymbol{A}$-concept (X, Y), $X \backslash \boldsymbol{A} = Y$ and $\boldsymbol{A} / Y = X$, the *conceptual image* $(\boldsymbol{F}/Y, X \backslash \boldsymbol{G})$ is an A-indexed collective $\boldsymbol{B}$-concept, since $\boldsymbol{B}/(X \backslash \boldsymbol{G}) = \boldsymbol{F}/Y$ and $(\boldsymbol{F}/Y) \backslash \boldsymbol{B} = X \backslash \boldsymbol{G}$. An important special case is the $L(\boldsymbol{A})$-indexed collective $\boldsymbol{A}$-concept (ι_A, τ_A). To state that the $\langle \boldsymbol{F}, \boldsymbol{G} \rangle$-image $(\boldsymbol{F}/\tau_A, \iota_A \backslash \boldsymbol{F})$ is an $L(\boldsymbol{A})$-indexed collective $\boldsymbol{B}$-concept, is to assert the pairing constraints $\boldsymbol{F}/\tau_A = \boldsymbol{B}/(\iota_A \backslash \boldsymbol{G})$ and $\iota_A \backslash \boldsymbol{G} = (\boldsymbol{F}/\tau_B) \backslash \boldsymbol{B}$. So, the concise definition in terms of pairing constraints, the original pointwise definition above, and the assertion that $\langle \boldsymbol{F}, \boldsymbol{G} \rangle$ preserves all collective concepts, are equivalent versions of the notion of a bonding pair.

```
(6) (CNG$function conceptual-image)
    (CNG$signature conceptual-image bonding-pair CNG$function)
    (forall (?bp (bonding-pair ?bp))
        (and (CNG$signature (conceptual-image ?bp)
                 (CLS.FIB$concept (source ?bp)) (CLS.FIB$concept (target ?bp)))
             (forall (?c ((CLS.FIB$concept (source ?bp)) ?c))
                 (and (= ((CLS.FIB$index (target ?bp)) ((conceptual-image ?bp) ?c))
                         ((CLS.FIB$index (source ?bp)) ?c))
                      (= ((CLS.FIB$extent (target ?bp)) ((conceptual-image ?bp) ?c))
                         ((CLS.FIB$right-derivation
                             (CLS.BND$classification (direct ?bp)))
                             ((CLS.FIB$intent (source ?bp)) ?c)))
                      (= ((CLS.FIB$intent (target ?bp)) ((conceptual-image ?bp) ?c))
                         ((CLS.FIB$left-derivation
                             (CLS.BND$classification (reverse ?bp)))
                             ((CLS.FIB$extent (source ?bp)) ?c)))))))
```

- Let $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \multimap \boldsymbol{B}$ and $\langle \boldsymbol{M}, \boldsymbol{N} \rangle : \boldsymbol{B} \multimap \boldsymbol{C}$ be two bonding pairs. Define the bonding pair composition $\langle \boldsymbol{F}, \boldsymbol{G} \rangle \circ \langle \boldsymbol{M}, \boldsymbol{N} \rangle \triangleq \langle \boldsymbol{F} \circ \boldsymbol{M}, \boldsymbol{N} \circ \boldsymbol{G} \rangle : \boldsymbol{A} \multimap \boldsymbol{C}$ in terms of bond composition.

```
(7) (CNG$function composition)
    (CNG$signature composition bonding-pair bonding-pair bonding-pair)
    (forall (?bp1 (bonding-pair ?bp1) ?bp2 (bonding-pair ?bp2))
        (<=> (exists (?bp (bonding-pair ?bp)) (= (composition ?bp1 ?bp2) ?bp))
             (= (target ?bp1) (source ?bp2))))
    (forall (?bp1 (bonding-pair ?bp1) ?bp2 (bonding-pair ?bp2))
        (=> (= (target ?bp1) (source ?bp2))
            (and (= (source (composition ?bp1 ?bp2)) (source ?bp1))
                 (= (target (composition ?bp1 ?bp2)) (target ?bp2))
                 (= (forward (composition ?bp1 ?bp2))
                    (CLS.BND$composition (forward ?bp1) (forward ?bp2)))
                 (= (reverse (composition ?bp1 ?bp2))
                    (CLS.BND$composition (reverse ?bp2) (reverse ?bp1))))))
```

- For any classification $\boldsymbol{A}$, define the bonding pair identity $\langle Id_A, Id_A \rangle : \boldsymbol{A} \multimap \boldsymbol{A}$ in terms of bond identity.

```
(8) (CNG$function identity)
    (CNG$signature identity CLS$classification bonding-pair)
    (forall (?a (CLS$classification ?a))
        (and (= (source (identity ?a)) ?a)
             (= (target (identity ?a)) ?a)
             (= (forward (identity ?a)) (CLS.BND$identity ?a))
             (= (reverse (identity ?a)) (CLS.BND$identity ?a))))
```

- For any classification $\boldsymbol{A}$ the type and instance embedding relations form bonding pairs in two different ways, $\langle \tau_A, \iota_A \rangle : \boldsymbol{A} \multimap L(\boldsymbol{A}) = A^2 \circ B^2(\boldsymbol{A})$ and $\langle \iota_A, \tau_A \rangle : A^2 \circ B^2(\boldsymbol{A}) = L(\boldsymbol{A}) \multimap \boldsymbol{A}$.

```
(9) (CNG$function tau-iota)
    (CNG$signature tau-iota CLS$classification bonding-pair)
    (forall (?a (CLS$classification ?a))
        (and (= (source (tau-iota ?a)) ?a)
```

```
            (= (target (tau-iota ?a))
               (CL$classification (CLS.CL$concept-lattice ?a)))
            (= (forward (tau-iota ?a)) (CLS.BND$tau ?a))
            (= (reverse (tau-iota ?a)) (CLS.BND$iota ?a))))

(10) (CNG$function iota-tau)
     (CNG$signature iota-tau CLS$classification bonding-pair)
      (forall (?a (CLS$classification ?a))
          (and (= (source (iota-tau ?a))
                  (CL$classification (CLS.CL$concept-lattice ?a)))
               (= (target (iota-tau ?a)) ?a)
               (= (forward (iota-tau ?a)) (CLS.BND$iota ?a))
               (= (reverse (iota-tau ?a)) (CLS.BND$tau ?a))))
```

- For any classification $\boldsymbol{A}$ the two bonding pairs, $\langle \tau_A, \iota_A \rangle : \boldsymbol{A} \multimap L(\boldsymbol{A}) = A^2 \circ B^2(\boldsymbol{A})$ and $\langle \iota_A, \tau_A \rangle : A^2 \circ B^2(\boldsymbol{A}) = L(\boldsymbol{A}) \multimap \boldsymbol{A}$, are inverse to each other: $\langle \tau_A, \iota_A \rangle \circ \langle \iota_A, \tau_A \rangle = Id_A$ and $\langle \iota_A, \tau_A \rangle \circ \langle \tau_A, \iota_A \rangle = Id_{L(A)}$. Therefore, each classification is isomorphic in the quasi-category Bonding Pair to its concept lattice: $\boldsymbol{A} \cong L(\boldsymbol{A}) = A^2 \circ B^2(\boldsymbol{A})$.

```
     (forall (?a (CLS$classification ?a))
         (and (= (composition (tau-iota ?a) (iota-tau ?a))
                 (identity ?a))
              (= (composition (iota-tau ?a) (tau-iota ?a))
                 (identity (CL$classification (CLS.CL$concept-lattice ?a))))))
```

- The functor composition $A^2 \circ B^2$ is naturally isomorphic to the identity functor $Id_{\text{Bonding Pair}}$. Let $\langle F, G \rangle : A \multimap B$ be a bonding pair. As shown above, the naturality conditions for bonds F and G are expressed as $\iota_A \circ F \circ \tau_B = A \circ B(F)$ and $\iota_B \circ G \circ \tau_A = A \circ B(G)$. So $\langle \iota_A, \tau_A \rangle \circ \langle F, G \rangle \circ \langle \tau_A, \iota_A \rangle = \langle \iota_A \circ F \circ \tau_B, \iota_B \circ G \circ \tau_A \rangle = \langle A \circ B(F), A \circ B(G) \rangle = A^2 \circ B^2(\langle F, G \rangle)$.

```
     (forall (?bp (bonding-pair ?bp))
         (= (composition (iota-tau (source ?bp)) ?bp)
            (composition
                (LAT.MOR$bonding-pair (CLS.BNDPR$homomorphism ?bp))
                (iota-tau (target ?bp)))))
```

- For any given bonding pair $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \multimap \boldsymbol{B}$ the *dual bonding pair* $\langle \boldsymbol{G}^{\text{op}}, \boldsymbol{F}^{\text{op}} \rangle : \boldsymbol{A}^{\text{op}} \multimap \boldsymbol{B}^{\text{op}}$ is the bonding pair with source/target classifications dualized, and forward/reverse bonds switched and dualized.

```
(11) (CNG$function opposite)
     (CNG$signature opposite bonding-pair bonding-pair)
     (forall (?bp (bonding-pair ?bp))
         (and (= (source (opposite ?bp)) (CLS$opposite (source ?bp)))
              (= (target (opposite ?bp)) (CLS$opposite (target ?bp)))
              (= (forward (opposite ?bp)) (CLS.BND$opposite (reverse ?bp)))
              (= (reverse (opposite ?bp)) (CLS.BND$opposite (forward ?bp)))))
```

- Let $\langle \boldsymbol{F}, \boldsymbol{G} \rangle : \boldsymbol{A} \multimap \boldsymbol{B}$ be any bonding pair. Then $\boldsymbol{F} : \boldsymbol{A} \multimap \boldsymbol{B}$ is a bond in the forward direction from classification $\boldsymbol{A}$ to classification $\boldsymbol{B}$, and $\boldsymbol{G} : \boldsymbol{A} \multimap \boldsymbol{B}$ is a bond in the reverse direction to classification $\boldsymbol{A}$ from classification $\boldsymbol{B}$. Applying the complete adjoint functor $A : \text{Bond} \to \text{Complete Adjoint}$, we get two adjoint pairs in opposite directions: an adjoint pair $\langle \varphi_F, \psi_F \rangle : L(\boldsymbol{B}) \rightleftarrows L(\boldsymbol{A})$ in the forward direction and an adjoint pair $\langle \varphi_G, \psi_G \rangle : L(\boldsymbol{A}) \rightleftarrows L(\boldsymbol{B})$ in the reverse direction. It was shown above that for bonding pairs the meet-preserving monotonic function $\psi_F : L(\boldsymbol{A}) \to L(\boldsymbol{B})$ is equal to the join-preserving monotonic function $\varphi_G : L(\boldsymbol{A}) \to L(\boldsymbol{B})$, giving a complete lattice homomorphism.

```
(12) (CNG$function homomorphism)
     (CNG$signature homomorphism bonding-pair CL.MOR$homomorphism)
     (forall (?bp (bonding-pair ?bp))
         (and (= (CL.MOR$source (homomorphism ?bp))
                 (CLS.CL$complete-lattice (source ?a)))
              (= (CL.MOR$target (homomorphism ?bp))
                 (CLS.CL$complete-lattice (target ?a)))
              (= (CL.MOR$forward (homomorphism ?bp))
                 (CLS.BND$adjoint-pair (forward ?h)))
              (= (CL.MOR$reverse (homomorphism ?bp))
                 (CLS.BND$adjoint-pair (reverse ?h)))))
```

This function is the unique mediating function for the $\mathsf{L}(A)$-indexed collective B-concept $(F/\tau_A, \iota_A \backslash G)$, the $\langle F, G \rangle$-image of the $\mathsf{L}(A)$-indexed collective A-concept (ι_A, τ_A), whose closure expressions define the pairing constraints.

Finite Colimits

CLS.COL

Classifications can be fused together and internalized using colimit operations. Here we present axioms that make CLASSIFICATION, the quasi-category of classifications and infomorphisms, finitely cocomplete. We assert the existence of initial classifications, binary coproducts of classifications, coequalizers of parallel pairs of infomorphisms and pushouts of spans of infomorphisms. Because of commonality, the terminology for binary coproducts, coequalizers and pushouts are put into sub-namespaces. The *diagrams*

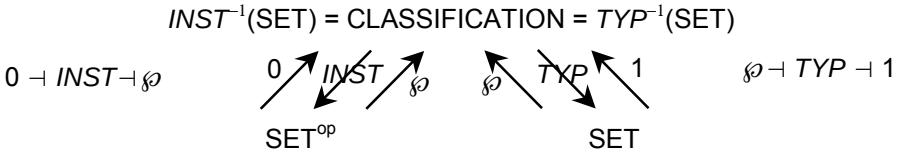


Figure 1: Fibered Span

and *colimits* are denoted by both generic and specific terminology.

The following discussion refers to Figure 1 (where arrows denote functors).

The existence of colimits is mediated through the quasi-adjunction $INST \dashv \wp$ between

$INST$: CLASSIFICATION $\to$ SET^{op} the underlying instance quasi-functor, and

$\wp$: SET^{op} $\to$ CLASSIFICATION the instance power quasi-functor (see Figure 1),

and the (trivial) quasi-adjunction $TYP \dashv 1$ between

TYP : CLASSIFICATION $\to$ SET the underlying type quasi-functor, and

1 : SET $\to$ CLASSIFICATION the (trivial) terminal type quasi-functor (see Figure 1).

Since the quasi-functors *INST* and *TYP* are left adjoint, they preserve all colimits. Using preservation as a guide, all diagrams, cocones and colimits in CLASSIFICATION have (and use) underlying instance/type diagrams, instance cones, type cocones, instance limits and type colimits in SET.

The Initial Classification

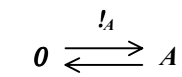


Figure 2: Initial Classification & Universality

- There is a special classification $\mathbf{0} = \langle 1 = \{0\}, \varnothing, \varnothing \rangle$ (see Figure 2, where arrows denote functions) called the *initial classification*, which has only one instance 0, no types, and empty incidence. The initial classification has the property that for any classification $\boldsymbol{A} = \langle inst(\boldsymbol{A}), typ(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$ there is a *counique infomorphism* $!_{\boldsymbol{A}} : \mathbf{0} \rightleftarrows \boldsymbol{A}$, the one and only infomorphism from $\mathbf{0}$ to $\boldsymbol{A}$. The instance function of this infomorphism is the unique function (the constant 0 function) from the instance class $inst(\boldsymbol{A})$ to the terminal class *1*. The type function of this infomorphism is the counique function (the empty function) from the initial (empty or null) class $\varnothing$ to the type class $typ(\boldsymbol{A})$. The fundamental constraint for the counique infomorphism is vacuous. Of course, this needs to be verified.

```
(1) (CLS$classification initial)
    (= (CLS$instance initial) SET.LIM$terminal)
    (= (CLS$type initial) SET.COL$initial)
    (= (CLS$incidence initial) SET.COL$empty)

(2) (CNG$function counique)
    (CNG$signature counique CLS$classification CLS.INFO$infomorphism)
    (forall (?a (CLS$classification ?a))
        (and (= (CLS.INFO$source (counique ?a)) initial)
             (= (CLS.INFO$target (counique ?a)) ?a)
             (= (CLS.INFO$instance (counique ?a)) (SET.LIM$unique (CLS$instance ?s)))
             (= (CLS.INFO$type (counique ?a)) (SET.COL$counique (CLS$type ?a)))))
```

Binary Coproducts

CLS.COL.COPRD

A *binary coproduct* is a finite colimit for a diagram of shape • •. Such a diagram (of classifications and infomorphisms) is called a *pair* of classifications. Given a pair of classifications $\boldsymbol{A}_1 = \langle \mathit{inst}(\boldsymbol{A}_1), \mathit{typ}(\boldsymbol{A}_1), \models_1 \rangle$ and $\boldsymbol{A}_2 = \langle \mathit{inst}(\boldsymbol{A}_2), \mathit{typ}(\boldsymbol{A}_2), \models_2 \rangle$, the *coproduct* or *sum* $\boldsymbol{A}_1 + \boldsymbol{A}_2$ (see top of Figure 3, where arrows denote functions) is the classification defined as follows:

- The set of instances is the Cartesian product $\mathit{inst}(\boldsymbol{A}_1+\boldsymbol{A}_2) = \mathit{inst}(\boldsymbol{A}_1)\times\mathit{inst}(\boldsymbol{A}_2)$. So, instances of $\boldsymbol{A}_1 + \boldsymbol{A}_2$ are pairs (i_1, i_2) of instances $i_1 \in \mathit{inst}(\boldsymbol{A}_1)$ and $i_2 \in \mathit{inst}(\boldsymbol{A}_2)$.
- The set of types is the disjoint union $\mathit{typ}(\boldsymbol{A}_1+\boldsymbol{A}_2) = \mathit{typ}(\boldsymbol{A}_1)+\mathit{typ}(\boldsymbol{A}_2)$. Concretely, types of $\boldsymbol{A}_1 + \boldsymbol{A}_2$ are either pairs $(1, t_1)$ where $t_1 \in \mathit{typ}(\boldsymbol{A}_1)$ or pairs $(2, t_2)$ where $t_2 \in \mathit{inst}(\boldsymbol{A}_2)$.
- The incidence $\models_{1+2}$ of $\boldsymbol{A}_1 + \boldsymbol{A}_2$ is defined by

  $(i_1, i_2) \models_{1+2} (1, t_1)$ when $i_1 \models_1 t_1$

  $(i_1, i_2) \models_{1+2} (2, t_2)$ when $i_2 \models_2 t_2$.

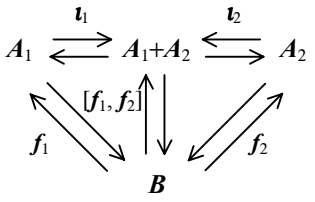


Figure 3: Binary Coproduct & Universality

- A *pair* (of classifications) is the appropriate base diagram for a binary coproduct. Each pair consists of a pair of classifications called *classification1* and *classification2*. Let either '`diagram`' or '`pair`' be the CLS namespace term that denotes the *Pair* collection. Pairs are determined by their two component classifications.

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

(2) (CNG$function classification1)
    (CNG$signature classification1 diagram CLS$classification)

(3) (CNG$function classification2)
    (CNG$signature classification2 diagram CLS$classification)

    (forall (?p (diagram ?p) ?q (diagram ?q))
        (=> (and (= (classification1 ?p) (classification1 ?q))
                 (= (classification2 ?p) (classification2 ?q)))
            (= ?p ?q)))
```

- There is an *instance pair* or *instance diagram* function, which maps a pair of classifications to the underlying pair of instance classes. Similarly, there is a *type pair* function, which maps a pair of classifications to the underlying pair of type classes.

```
(4) (CNG$function instance-diagram)
    (CNG$function instance-pair)
    (= instance-pair instance-diagram)
    (CNG$signature instance-diagram diagram SET.LIM.PRD$diagram)
    (forall (?p (diagram ?p))
        (and (= (SET.LIM.PRD$class1 (instance-diagram ?p))
                (CLS$instance (classification1 ?p)))
             (= (SET.LIM.PRD$class2 (instance-diagram ?p))
                (CLS$instance (classification2 ?p)))))

(5) (CNG$function type-diagram)
    (CNG$function type-pair)
    (= type-pair type-diagram)
    (CNG$signature type-diagram diagram SET.COLIM.COPRD$diagram)
    (forall (?p (diagram ?p))
        (and (= (SET.COLIM.COPRD$class1 (type-diagram ?p))
                (CLS$type (classification1 ?p)))
             (= (SET.COLIM.COPRD$class2 (type-diagram ?p))
                (CLS$type (classification2 ?p)))))
```

- Every pair has an *opposite*.

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

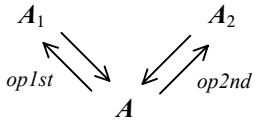
```
(forall (?p (pair ?p))
    (and (= (classification1 (opposite ?p)) (classification2 ?p))
         (= (classification2 (opposite ?p)) (classification1 ?p))))
```

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

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

- A *coproduct cocone* is the appropriate cocone for a binary coproduct. A coproduct cocone (Figure 4, where arrows denote functions) consists of a pair of infomorphisms called *opfirst* and *opsecond*. These are required to have a common source class called the *opvertex* of the cocone. Each coproduct cocone is over a pair. A coproduct cocone is the very special case of a cocone over a pair (of classifications). Let 'cocone' be the CLS term that denotes the *Coproduct Cocone* collection.

Figure 4: Coproduct Cocone



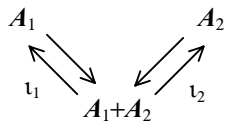

```
(7) (CNG$conglomerate cocone)

(8) (CNG$function cocone-diagram)
    (CNG$signature cocone-diagram cocone diagram)

(9) (CNG$function opvertex)
    (CNG$signature opvertex cocone CLS$classification)

(10) (CNG$function opfirst)
     (CNG$signature opfirst cocone CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
         (and (= (CLS.INFO$source (opfirst ?s))
                 (classification1 (cocone-diagram ?s)))
              (= (CLS.INFO$target (opfirst ?s)) (opvertex ?s))))

(11) (CNG$function opsecond)
     (CNG$signature opsecond cocone CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
         (and (= (CLS.INFO$source (opsecond ?s))
                 (classification2 (cocone-diagram ?s)))
              (= (CLS.INFO$target (opsecond ?s)) (opvertex ?s))))
```

- There is an *instance cone* function, which maps a coproduct cocone of classifications and infomorphisms to the underlying product cone of instance classes and instance functions. Similarly, there is a *type cocone* function, which maps a coproduct cocone of classifications and infomorphisms to the underlying cocone of type classes and type functions.

```
(12) (CNG$function instance-cone)
     (CNG$signature instance-cone cocone SET.LIM.PRD$cone)
     (forall (?s (cocone ?s))
         (and (= (SET.LIM.PRD$cone-diagram (instance-cone ?s))
                 (instance-diagram (cocone-diagram ?s)))
              (= (SET.LIM.PRD$vertex (instance-cone ?s))
                 (CLS$instance (opvertex ?s)))
              (= (SET.LIM.PRD$first (instance-pair ?s))
                 (CLS.INFO$instance (opfirst ?s)))
              (= (SET.LIM.PRD$second (instance-pair ?s))
                 (CLS.INFO$instance (opsecond ?s)))))

(13) (CNG$function type-cocone)
     (CNG$signature type-cocone cocone SET.COLIM.COPRD$cocone)
     (forall (?s (cocone ?s))
         (and (= (SET.COLIM.COPRD$cocone-diagram (type-cocone ?s))
                 (type-diagram (cocone-diagram ?s)))
              (= (SET.COLIM.COPRD$opvertex (type-cocone ?s))
                 (CLS$type (opvertex ?s)))
              (= (SET.COLIM.COPRD$opfirst (type-cocone ?s))
                 (CLS.INFO$type (opfirst ?s)))
              (= (SET.COLIM.COPRD$opsecond (type-cocone ?s))
                 (CLS.INFO$type (opsecond ?s)))))
```

- There is a unary CNG function '`colimiting-cone`' that maps a pair (of classifications) to its binary coproduct (colimiting binary coproduct cocone) (Figure 5, where arrows denote functions). Axiom (*) asserts that this function is total. This, along with the universality of the comediator infomorphism, implies that a binary coproduct exists for any pair of classifications. The opvertex of the colimiting binary coproduct cocone is a specific *Binary Coproduct* class given by the CNG function '`binary-coproduct`'. It comes equipped with two CNG injection infomorphisms '`injection1`' and '`injection2`'. This notation is for convenience of reference. It is used for pushouts in general. Axiom (#) is the necessary condition that the instance and type quasi-functors preserve concrete colimits. This ensures that both this coproduct and its injection infomorphisms are specific – that its instance class is exactly the Cartesian product of the instance classes of the pair of classifications and that its type class is exactly the disjoint union of the type classes of the pair of classifications. The injection terms in axioms (16–17) are created for convenience of reference. That these are infomorphisms needs to be checked; that is, the instance and types of the source and target classifications need to be checked for correctness, and the fundamental property for these infomorphisms must be verified.

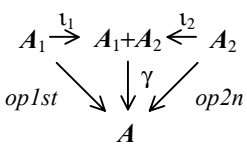
Figure 5: Colimiting Cocone




```
(14) (CNG$function colimiting-cocone)
     (CNG$signature colimiting-cocone diagram cocone)
 (*) (forall (?p (diagram ?p))
         (exists (?s (cocone ?s))
             (= (colimiting-cocone ?p) ?s)))

 (#) (forall (?p (diagram ?p))
         (and (= (cocone-diagram (colimiting-cocone ?p)) ?p)
              (= (instance-cone (colimiting-cocone ?p))
                 (SET.LIM.PRD$limiting-cone (instance-diagram ?p))
              (= (type-cocone (colimiting-cocone ?p))
                 (SET.COL.COPRD$colimiting-cocone (type-diagram ?p))))))

(15) (CNG$function colimit)
     (CNG$function binary-coproduct)
     (= binary-coproduct colimit)
     (CNG$signature colimit diagram CLS$classification)
     (forall (?p (diagram ?p))
         (= (colimit ?p) (opvertex (colimiting-cocone ?p))))

(16) (CNG$function injection1)
     (CNG$signature injection1 diagram CLS.INFO$infomorphism)
     (forall (?p (diagram ?p))
         (and (= (CLS.INFO$source (injection1 ?p)) (classification1 ?p))
              (= (CLS.INFO$target (injection1 ?p)) (colimit ?p))
              (= (injection1 ?p) (opfirst (colimiting-cocone ?p))))))

(17) (CNG$function injection2)
     (CNG$signature injection2 diagram CLS.INFO$infomorphism)
     (forall (?p (diagram ?p))
         (and (= (CLS.INFO$source (injection2 ?p)) (classification2 ?p))
              (= (CLS.INFO$target (injection2 ?p)) (colimit ?p))
              (= (injection2 ?p) (opsecond (colimiting-cocone ?p))))))

 (#) (forall (?p (diagram ?p))
         (forall (?i ((CLS$instance (colimit ?p)) ?i)
                  ?t ((CLS$type (colimit ?p)) ?t))
             (<=> (= ((CLS$incidence (colimit ?p)) [?i ?t])
                  (or ((CLS$incidence (classification1 ?p)) [(?i 1) (?t 2)])
                      ((CLS$incidence (classification2 ?p)) [(?i 2) (?t 2)])))))))
```

- For any cocone, there is a *comediator* infomorphism (see Figure 6, where arrows denote infomorphisms) from the binary coproduct of the underlying diagram (pair of classifications) to the opvertex of the cocone. This is the unique infomorphism, which commutes with opfirst and opsecond. We use a KIF definite description abbreviation to define this. Existence and uniqueness represents the universality of the binary coproduct operator. That

Figure 6: Coproduct Comediator



this is an infomorphism needs to be checked; that is, the instance and types of the source and target classifications need to be checked for correctness, and the fundamental property for this infomorphism must be verified.

```
(18) (CNG$function comediator)
     (CNG$signature comediator cocone CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
         (and (= (CLS.INFO$source (comediator ?s)) (colimit (cocone-diagram ?s)))
              (= (CLS.INFO$target (comediator ?s)) (opvertex ?s))
              (= (CLS.INFO$instance (comediator ?s))
                 (SET.LIM.PRD$mediator (instance-cone ?s)))
              (= (CLS.INFO$type (comediator ?s))
                 (SET.COL.COPRD$comediator (type-cocone ?s)))))
```

- It can also be verified that the comediator is the unique infomorphism that makes the diagram in Figure 4 commutative.

```
(forall (?s (cocone ?s))
    (= (comediator ?s)
       (the (?f (CLS.INFO$infomorphism ?f))
           (and (= (CLS.INFO$composition (injection1 (cocone-diagram ?s)) ?f)
                   (opfirst ?s))
                (= (CLS.INFO$composition (injection2 (cocone-diagram ?s)) ?f)
                   (opsecond ?s))))))
```

Coinvariants and Coquotients

CLS.COL.COINV

- Given a classification $\boldsymbol{A} = \langle \mathit{inst}(\boldsymbol{A}), \mathit{typ}(\boldsymbol{A}), \models_{\boldsymbol{A}} \rangle$, a *coinvariant* (called a *dual invariant* in Barwise and Seligman, 1997) is a pair $J = \langle A, R \rangle$ consisting of a subclass of instances $A \subseteq \mathit{inst}(\boldsymbol{A})$ and a binary endorelation $\boldsymbol{R}$ on types $R \subseteq \mathit{typ}(\boldsymbol{A}) \times \mathit{typ}(\boldsymbol{A})$ that satisfies the fundamental constraint:

  if $(t_0, t_1) \in R$ then for each $i \in A$, $i \models_{\boldsymbol{A}} t_0$ iff $i \models_{\boldsymbol{A}} t_1$.

  The classification $\boldsymbol{A}$ is called the *base classification* of J – the classification on which J is based.

```
(1) (CNG$conglomerate coinvariant)

(2) (CNG$function classification)
    (CNG$function base)
    (= base classification)
    (CNG$signature base coinvariant CLS$classification)

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

(4) (CNG$function endorelation)
    (CNG$signature endorelation coinvariant REL.ENDO$endorelation)

    (forall (?j (coinvariant ?j))
        (and (SET$subclass (class ?j) (CLS$instance (base ?j)))
             (SET$subclass (REL.ENDO$class (endorelation ?j)) (CLS$type (base ?j)))
             (forall (?i ((class ?j) ?i)
                         ?t0 ?t1 (REL.ENDO$extent (endorelation ?j) [?t0 ?t1]))
                 (<=> ((CLS$incidence (base ?j)) [?i ?t0])
                      ((CLS$incidence (base ?j)) [?i ?t1])))))
```

- Often, the relation $\boldsymbol{R}$ is an equivalence relation on the types. However, it is convenient not to require this. The endorelation $\boldsymbol{R}$ is contained in a smallest equivalence relation $\equiv_{\boldsymbol{R}}$ on types called the equivalence relation generated by $\boldsymbol{R}$. This is the reflexive, symmetric, transitive closure of $\boldsymbol{R}$. For any type $t \in \mathit{typ}(\boldsymbol{A})$, write $[t]_{\boldsymbol{R}}$ for the $\boldsymbol{R}$-equivalence class of t. The *coquotient* (called the *dual quotient* in Barwise and Seligman, 1997) $\boldsymbol{A}/J$ of a coinvariant J on a classification $\boldsymbol{A}$ (see Figure 7, where arrows denote functions) is the classification defined as follows:
  - The set of instances is A, the given subset of $\mathit{inst}(\boldsymbol{A})$.
  - The set of types is $\mathit{typ}(\boldsymbol{A})/\boldsymbol{R}$, the set of $\boldsymbol{R}$-equivalence classes of $\mathit{typ}(\boldsymbol{A})$.

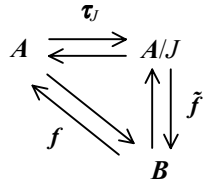


Figure 7: Universality for the Coquotient

– The incidence $\models_{A/J}$ of $\boldsymbol{A}/J$ is defined by

$i \models_{A/J} [t]_R$ when $i \models_A t$

which is well-defined by the fundamental constraint.

There is a *canonical quotient infomorphism* $\boldsymbol{\tau}_J : \boldsymbol{A} \rightleftarrows \boldsymbol{A}/J$, whose instance function is the inclusion function $inc : A \to \mathit{inst}(\boldsymbol{A})$, and whose type function is the canonical quotient function and $[-]_{\boldsymbol{R}} : \mathit{typ}(\boldsymbol{A}) \to \mathit{typ}(\boldsymbol{A})/\boldsymbol{R}$. The fundamental property for this infomorphism is trivial, given the definition of the coquotient incidence above.

Here is the KIF formalism for coinvariants and coquotients. A coinvariant is determined by its base, class and endorelation triple.

```
(5) (CNG$function coquotient)
    (CNG$signature coquotient coinvariant CLS$classification)
    (forall (?j (coinvariant ?j))
        (and (= (CLS$instance (coquotient ?j))
                (class ?j))
             (= (CLS$type (coquotient ?j))
                (REL.ENDO$quotient (REL.ENDO$equivalence-closure (endorelation ?j))))
             (forall (?i ((class ?j) ?i)
                      ?t ((CLS$type (base ?j)) ?t))
                 (<=> ((CLS$incidence (coquotient ?j))
                           [?i ((REL.ENDO$canon (endorelation ?j)) ?t)])
                      ((CLS$incidence (base ?j)) [?i ?t]))))))

(6) (CNG$function canon)
    (CNG$signature canon coinvariant CLS.INFO$infomorphism)
    (forall (?j (coinvariant ?j))
        (and (= (CLS.INFO$source (canon ?j)) (base ?j))
             (= (CLS.INFO$target (canon ?j)) (coquotient ?j))
             (= (CLS.INFO$instance (canon ?j))
                (SET.FTN$inclusion (class ?j) (CLS$instance (base ?j))))
             (= (CLS.INFO$type (canon ?j))
                (REL.ENDO$canon (endorelation ?j))))))
```

Let $J = \langle A, R \rangle$ be a coinvariant on a classification $\boldsymbol{A}$. An infomorphism $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ *respects* J when:

1. For any instance $i \in \mathit{inst}(\boldsymbol{B})$, $\mathit{inst}(\boldsymbol{f})(i) \in A$; and
2. For any two types $t_0, t_1 \in \mathit{typ}(\boldsymbol{A})$, if $(t_0, t_1) \in R$ then $\mathit{typ}(\boldsymbol{f})(t_0) = \mathit{typ}(\boldsymbol{f})(t_1)$.

```
(7) (CNG$relation respects)
    (CNG$signature respects CLS.INFO$infomorphism coinvariant)
    (forall (?j (coinvariant ?j)
             ?f (CLS.INFO$infomorphism ?f))
        (<=> (respects ?f ?j)
             (and (= (CLS.INFO$source ?f) (base ?j)))
                  (forall (?i ((CLS.INFO$target ?f) ?i))
                      ((class ?j) ((CLS.INFO$instance ?f) ?i)))
                  (forall (?t0 ((CLS.INFO$source ?f) ?t0)
                           ?t1 ((CLS.INFO$source ?f) ?t1))
                      (=> ((REL.ENDO$extent (endorelation ?j)) [?t0 ?t1])
                          (= ((CLS.INFO$type ?f) ?t0) ((CLS.INFO$type ?f) ?t1))))))
```

Proposition. *Let J be a coinvariant on a classification $\boldsymbol{A}$. For every infomorphism $\boldsymbol{f} : \boldsymbol{A} \rightleftarrows \boldsymbol{B}$ that respects J, there is a unique comediating infomorphism $\tilde{\boldsymbol{f}} : \boldsymbol{A}/J \rightleftarrows \boldsymbol{B}$ such that $\boldsymbol{\tau}_J \circ \tilde{\boldsymbol{f}} = \boldsymbol{f}$ (the diagram in Figure 5 commutes).*

This proposition is used to define a *comediator* function, which maps a respectful infomorphism to its unique associate.

```
(8) (KIF$function comediator)
    (KIF$signature comediator coinvariant CNG$function)
    (forall (?j (coinvariant ?j)
        (and (CNG$signature (comediator ?j)
                 CLS.INFO$infomorphism CLS.INFO$infomorphism)
             (forall (?f (CLS.INFO$infomorphism ?f) (respects ?f ?j))
```

```
(= ((comediator ?j) ?f)
   (the (?ft (CLS.INFO$infomorphism ?ft))
        (and (= (CLS.INFO$source ?ft) (coquotient ?j))
             (= (CLS.INFO$target ?ft) (CLS.INFO$target ?f))
             (= (CLS.INFO$composition (coquotient ?j) ?ft) ?f)))))))
```

Coequalizers

CLS.COL.COEQ

A (binary) *coequalizer* is a finite colimit in CLASSIFICATION 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* (see Figure 8, where arrows denote infomorphisms) is the appropriate base diagram for a coequalizer. Each parallel pair consists of a pair of infomorphisms called *infomorphism1* and *infomorphism2* that share the same *source* and *target* classifications. Let either '`diagram`' or '`parallel-pair`' be the term that denotes the *Parallel Pair* collection. Parallel pairs are determined by their two component infomorphisms.

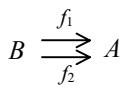


Figure 8: Parallel Pair

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

(2) (CNG$function source)
    (CNG$signature source diagram CLS$classification)

(3) (CNG$function target)
    (CNG$signature target diagram CLS$classification)

(4) (CNG$function infomorphism1)
    (CNG$signature infomorphism1 diagram CLS.INFO$infomorphism)

(5) (CNG$function infomorphism2)
    (CNG$signature infomorphism2 diagram CLS.INFO$infomorphism)

    (forall (?p (diagram ?p))
        (and (= (SET.FTN$source (infomorphism1 ?p)) (source ?p))
             (= (SET.FTN$target (infomorphism1 ?p)) (target ?p))
             (= (SET.FTN$source (infomorphism2 ?p)) (source ?p))
             (= (SET.FTN$target (infomorphism2 ?p)) (target ?p))))

    (forall (?p (diagram ?p) ?q (diagram ?q))
        (=> (and (= (infomorphism1 ?p) (infomorphism1 ?q))
                 (= (infomorphism2 ?p) (infomorphism2 ?q)))
            (= ?p ?q)))
```

- There is an *instance parallel pair* or *instance diagram* function, which maps a parallel pair of infomorphisms to the underlying (`SET.LIM.EQU`) parallel pair of instance functions. Similarly, there is a *type parallel pair* or *type diagram* function, which maps a parallel pair of infomorphisms to the underlying (`SET.COLIM.COEQ`) parallel pair of type functions.

```
(6) (CNG$function instance-diagram)
    (CNG$function instance-parallel-pair)
    (= instance-parallel-pair instance-diagram)
    (CNG$signature instance-diagram diagram SET.LIM.EQU$diagram)
    (forall (?p (diagram ?p))
        (and (= (SET.LIM.EQU$source (instance-diagram ?p))
                (CLS$instance (target ?p)))
             (= (SET.LIM.EQU$target (instance-diagram ?p))
                (CLS$instance (source ?p)))
             (= (SET.LIM.EQU$function1 (instance-diagram ?p))
                (CLS.INFO$instance (infomorphism1 ?p)))
             (= (SET.LIM.EQU$function2 (instance-diagram ?p))
                (CLS.INFO$instance (infomorphism2 ?p)))))

(7) (CNG$function type-diagram)
    (CNG$function type-parallel-pair)
    (= type-parallel-pair type-diagram)
    (CNG$signature type-diagram diagram SET.COLIM.COEQ$diagram)
```

```
(forall (?p (diagram ?p))
    (and (= (SET.COLIM.COEQ$source (type-diagram ?p))
            (CLS$type (source ?p)))
         (= (SET.COLIM.COEQ$target (type-diagram ?p))
            (CLS$type (target ?p)))
         (= (SET.COLIM.COEQ$function1 (type-diagram ?p))
            (CLS.INFO$type (infomorphism1 ?p)))
         (= (SET.COLIM.COEQ$function2 (type-diagram ?p))
            (CLS.INFO$type (infomorphism2 ?p)))))
```

- Every parallel pair of infomorphisms has an associated coinvariant whose base classification is the target of the parallel pair. There is a *coinvariant* function, which maps a parallel pair of infomorphisms to the associated coinvariant.

```
(8) (CNG$function coinvariant)
    (CNG$signature coinvariant diagram CLS.COL.COINV$coinvariant)
    (forall (?p (diagram ?p))
        (and (= (CLS.COL.COINV$base (coinvariant ?p))
                (target ?p))
             (= (CLS.COL.COINV$class (coinvariant ?p))
                (SET.LIM.EQU$equalizer (instance-diagram ?p)))
             (= (CLS.COL.COINV$endorelation (coinvariant ?p))
                (SET.COL.COEQ$endorelation (type-diagram ?p)))))
```

- *Coequalizer Cocones* are used to specify and axiomatize coequalizers. Each coequalizer cocone (see Figure 9, where arrows denote infomorphisms) has an underlying *parallel-pair*, an *opvertex* class, and an infomorphism called *infomorphism*, whose source classification is the target classification of the infomorphisms in the parallel-pair and whose target classification is the opvertex. The composite infomorphism indicated in the diagram below is obviously not needed. An coequalizer cocone is the very special case of a cocone over an parallel-pair. Let ‘cocone’ be the term that denotes the *Coequalizer Cocone* collection.

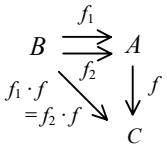


Figure 9: Coequalizer Cocone

```
(9) (CNG$conglomerate cocone)

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

(11) (CNG$function opvertex)
     (CNG$signature opvertex cocone CLS$classification)

(12) (CNG$function infomorphism)
     (CNG$signature infomorphism cocone CLS.INFO$infomorphism)

     (forall (?s (cocone ?s))
        (and (= (CLS.INFO$source (infomorphism ?s)) (target (cocone-diagram ?s)))
             (= (CLS.INFO$target (infomorphism ?s)) (opvertex ?s))
             (= (CLS.INFO$composition
                    (infomorphism1 (cocone-diagram ?s))
                    (infomorphism ?s))
                (CLS.INFO$composition
                    (infomorphism2 (cocone-diagram ?s))
                    (infomorphism ?s)))))
```

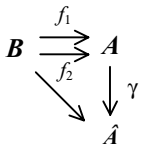
- There is an *instance equalizer cone* function, which maps a coequalizer cocone of classifications and infomorphisms to the underlying equalizer cone of instance classes and instance functions. Similarly, there is a *type coequalizer cocone* function, which maps a coequalizer cocone of classifications and infomorphisms to the underlying coequalizer cocone of type classes and type functions.

```
(13) (CNG$function instance-cone)
     (CNG$signature instance-cone cocone SET.LIM.EQU$cone)
     (forall (?s (cocone ?s))
         (and (= (SET.LIM.EQU$cone-diagram (instance-cone ?s))
                 (instance-diagram (cocone-diagram ?s)))
              (= (SET.LIM.EQU$vertex (instance-cone ?s))
                 (CLS$instance (opvertex ?s)))
              (= (SET.LIM.EQU$function (instance-pair ?s))
```

```
                (CLS.INFO$instance (infomorphism ?s)))))

(14) (CNG$function type-cocone)
     (CNG$signature type-cocone cocone SET.COLIM.COEQ$cocone)
     (forall (?s (cocone ?s))
         (and (= (SET.COLIM.COEQ$cocone-diagram (type-cocone ?s))
                 (type-diagram (cocone-diagram ?s)))
              (= (SET.COLIM.COEQ$opvertex (type-cocone ?s))
                 (CLS$type (opvertex ?s)))
              (= (SET.COLIM.COEQ$function (type-cocone ?s))
                 (CLS.INFO$type (infomorphism ?s)))))
```

- There is a unary CNG function '`limiting-cone`' that maps a parallel pair (of infomorphisms) to its coequalizer (colimiting coequalizer cocone) (see Figure 10, where arrows denote infomorphisms). Axiom (*) asserts that this function is total. This, along with the universality of the comediator infomorphism *gamma*, implies that a coequalizer exists for any parallel pair of infomorphisms. The opvertex of the colimiting coequalizer cocone is a specific *Coequalizer* class given by the CNG function '`coequalizer`'. It comes equipped with a CNG quotient infomorphism '`canon`'. This notation is for convenience of reference. Axiom (#) is the necessary condition that the instance and type quasi-functors preserve concrete colimits. This ensures that both this coequalizer and its canon infomorphism are specific. The canon term in axiom (17) is created for convenience of reference. That this is an infomorphism needs to be checked; that is, the instance and types of the source and target classifications need to be checked for correctness, and the fundamental property for this infomorphism must be verified.

Figure 10: Colimiting Cocone




```
(15) (CNG$function colimiting-cocone)
     (CNG$signature colimiting-cocone diagram cocone)
 (*) (forall (?p (diagram ?p))
         (exists (?s (cocone ?s))
             (= (colimiting-cocone ?p) ?s)))

 (#) (forall (?p (diagram ?p))
         (and (= (cocone-diagram (colimiting-cocone ?p)) ?p)
              (= (instance-cone (colimiting-cocone ?p))
                 (SET.LIM.EQU$limiting-cone (instance-diagram ?p))
              (= (type-cocone (colimiting-cocone ?p))
                 (SET.COL.COEQ$colimiting-cocone (type-diagram ?p))))))

(16) (CNG$function colimit)
     (CNG$function coequalizer)
     (= coequalizer colimit)
     (CNG$signature colimit diagram CLS$classification)
     (forall (?p (diagram ?p))
        (= (colimit ?p) (opvertex (colimiting-cocone ?p))))

(17) (CNG$function canon)
     (CNG$signature canon diagram CLS.INFO$infomorphism)
     (forall (?p (diagram ?p))
         (and (= (CLS.INFO$source (canon ?p)) (target ?p))
              (= (CLS.INFO$target (canon ?p)) (colimit ?p))
              (= (canon ?p) (infomorphism (colimiting-cocone ?p)))))

 (#) (forall (?p (diagram ?p))
         (and (SET.FTN$surjection (CLS.INFO$type (canon ?p)))
              (forall (?i ((class ?j) ?i)
                       ?t ((CLS$type (target ?p)) ?t))
                  (<=> ((CLS$incidence (colimit ?p))
                           [?i ((CLS.INFO$type (canon ?p)) ?t)])
                       ((CLS$incidence (target ?p)) [?i ?t])))
```

- The coquotient classification of the coinvariant of a coequalizer diagram (parallel pair of infomorphisms) is (not just isomorphic but) equal to the coequalizer classification. Likewise, the canon infomorphism of the coinvariant of a coequalizer diagram (parallel pair of infomorphisms) is equal to the coequalizer canon infomorphism.

```
(forall (?p (diagram ?p))
     (and (= (CLS.COL.COINV$coquotient (coinvariant ?p)) (coequalizer ?p))
          (= (CLS.COL.COINV$canoon (coinvariant ?p)) (canon ?p))))
```

Pushouts

CLS.COL.PSH

Given a pair of infomorphisms $\boldsymbol{f}_1 : \boldsymbol{A} \rightleftarrows \boldsymbol{B}_1$ and $\boldsymbol{f}_2 : \boldsymbol{A} \rightleftarrows \boldsymbol{B}_2$ with a common source classification $\boldsymbol{A}$, a *pushout* of $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ (see Figure 11, where arrows denote functions) consists of a classification $\boldsymbol{P}$ and a pair of infomorphisms $\boldsymbol{m}_1 : \boldsymbol{B}_1 \rightleftarrows \boldsymbol{P}$ and $\boldsymbol{m}_2 : \boldsymbol{B}_2 \rightleftarrows \boldsymbol{P}$ with common target classification $\boldsymbol{P}$, that satisfy the following universality conditions.

1. $\boldsymbol{f}_1 \circ \boldsymbol{m}_1 = \boldsymbol{f}_2 \circ \boldsymbol{m}_2$
2. if $\boldsymbol{C}$ is any classification and $\boldsymbol{g}_1 : \boldsymbol{B}_1 \rightleftarrows \boldsymbol{C}$ and $\boldsymbol{g}_2 : \boldsymbol{B}_2 \rightleftarrows \boldsymbol{C}$ are a pair of infomorphisms with the common target classification $\boldsymbol{C}$, that satisfy $\boldsymbol{f}_1 \circ \boldsymbol{g}_1 = \boldsymbol{f}_2 \circ \boldsymbol{g}_2$, then there is a unique infomorphism $\boldsymbol{g} : \boldsymbol{P} \rightleftarrows \boldsymbol{C}$ satisfying $\boldsymbol{m}_1 \circ \boldsymbol{g} = \boldsymbol{g}_1$ and $\boldsymbol{m}_2 \circ \boldsymbol{g} = \boldsymbol{g}_2$.

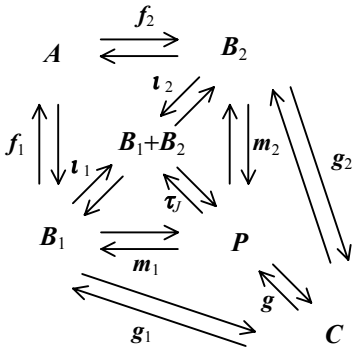


Figure 11: Pushout & Universality

In order to build the pushout we implicitly use a binary coproduct classification, a coinvariant on that coproduct, and a pushout classification that is the coquotient for that coinvariant. However, these elements are phrased in terms of a pullback of the instance component and a pushout of the type component of the infomorphism pair. Here is the definition in more detail.

1. Construct the Set pullback $\mathit{inst}(\boldsymbol{B}_1) \times_{\mathit{inst}(\boldsymbol{A})} \mathit{inst}(\boldsymbol{B}_2)$, whose vertex set is

   $A = \{(b_1, b_2) \mid b_1 \in \mathit{inst}(\boldsymbol{B}_1), b_2 \in \mathit{inst}(\boldsymbol{B}_2), \mathit{inst}(\boldsymbol{f}_1)(b_1) = \mathit{inst}(\boldsymbol{f}_2)(b_2)\}$
   = pullback vertex in the category Set of $\mathit{inst}(\boldsymbol{f}_1)$ and $\mathit{inst}(\boldsymbol{f}_2)$

   a subset of the binary Cartesian product $\mathit{inst}(\boldsymbol{B}_1) \times \mathit{inst}(\boldsymbol{B}_2)$. Let the pullback "projections" be denoted

   $\mathit{first} : \mathit{inst}(\boldsymbol{P}) \to \mathit{inst}(\boldsymbol{B}_1)$ and $\mathit{second} : \mathit{inst}(\boldsymbol{P}) \to \mathit{inst}(\boldsymbol{B}_2)$.

2. Construct the Set pushout $\mathit{typ}(\boldsymbol{B}_1) +_{\mathit{typ}(\boldsymbol{A})} \mathit{typ}(\boldsymbol{B}_2)$, whose endorelation is

   $R = \{((1, \mathit{typ}(\boldsymbol{f}_1)(t)), (2, \mathit{typ}(\boldsymbol{f}_2)(t))) \mid \text{some } t \in \mathit{typ}(\boldsymbol{A})\}$
   = pushout endorelation in the category Set of $\mathit{typ}(\boldsymbol{f}_1)$ and $\mathit{typ}(\boldsymbol{f}_2)$

   whose object set is the binary disjoint union $\mathit{typ}(\boldsymbol{B}_1) + \mathit{typ}(\boldsymbol{B}_2)$. Let the pushout "injections" be denoted

   $\mathit{opfirst} : \mathit{typ}(\boldsymbol{B}_1) \to \mathit{typ}(\boldsymbol{P})$ and $\mathit{opsecond} : \mathit{typ}(\boldsymbol{B}_2) \to \mathit{typ}(\boldsymbol{P})$.

3. Define the coinvariant $J = \langle A, R \rangle$ on the coproduct $\boldsymbol{B}_1 + \boldsymbol{B}_2$.

   Checked by:

   $(b_1, b_2) \vDash_{1+2} (1, \mathit{typ}(\boldsymbol{f}_1)(t))$ iff $b_1 \vDash_1 \mathit{typ}(\boldsymbol{f}_1)(t)$ iff $\mathit{inst}(\boldsymbol{f}_1)(b_1) \vDash_{\boldsymbol{A}} t$

   iff $\mathit{inst}(\boldsymbol{f}_2)(b_2) \vDash_{\boldsymbol{A}} t$ iff $b_2 \vDash_2 \mathit{typ}(\boldsymbol{f}_2)(t)$ iff $(b_1, b_2) \vDash_{1+2} (2, \mathit{typ}(\boldsymbol{f}_2)(t))$

4. Specify the pushout to be the associated coquotient.

   $\boldsymbol{P} = (\boldsymbol{B}_1 + \boldsymbol{B}_2)/J$
   $\boldsymbol{m}_1 = \langle \mathit{first}, \mathit{opfirst} \rangle = \boldsymbol{\iota}_1 \circ \boldsymbol{\tau}_J$
   $\boldsymbol{m}_2 = \langle \mathit{second}, \mathit{opsecond} \rangle = \boldsymbol{\iota}_2 \circ \boldsymbol{\tau}_J$

   Checked by:

   $\mathit{inst}(\boldsymbol{m}_1)(b_1, b_2) \vDash_1 t_1$ iff $b_1 \vDash_1 t_1$ iff $(b_1, b_2) \vDash_{1+2} (1, t_1)$ iff $(b_1, b_2) \vDash_{(1+2)/J} [(1, t_1)]_R$
   iff $(b_1, b_2) \vDash_{(1+2)/J} \mathit{inst}(\boldsymbol{m}_1)(t_1)$

We demonstrate universality.

1. The equality $typ(\boldsymbol{f}_1) \circ typ(\boldsymbol{m}_1) = typ(\boldsymbol{f}_2) \circ typ(\boldsymbol{m}_2)$ holds, since the type function of the canonical quotient infomorphism maps equivalent types in $typ(\boldsymbol{B}_1+\boldsymbol{B}_2)$ to the same type in $typ(\boldsymbol{P}) = typ((\boldsymbol{B}_1+\boldsymbol{B}_2)/J) = typ(\boldsymbol{B}_1+\boldsymbol{B}_2)/R$. The equality $inst(\boldsymbol{m}_1) \circ inst(\boldsymbol{f}_1) = inst(\boldsymbol{m}_2) \circ inst(\boldsymbol{f}_2)$ holds, since $(b_1, b_2) \in inst(\boldsymbol{P}) = A$ implies $inst(\boldsymbol{f}_1)(inst(\boldsymbol{m}_1)(b_1, b_2)) = inst(\boldsymbol{f}_1)(b_1) = inst(\boldsymbol{f}_2)(b_2) = inst(\boldsymbol{f}_2)(inst(\boldsymbol{m}_2)(b_1, b_2))$. So, $\boldsymbol{f}_1 \circ \boldsymbol{m}_1 = \boldsymbol{f}_2 \circ \boldsymbol{m}_2$.
2. Suppose $\boldsymbol{C}$ is any classification and $\boldsymbol{g}_1 : \boldsymbol{B}_1 \rightleftarrows \boldsymbol{C}$ and $\boldsymbol{g}_2 : \boldsymbol{B}_2 \rightleftarrows \boldsymbol{C}$ are a pair of infomorphisms with the common target classification $\boldsymbol{C}$, that satisfy $\boldsymbol{f}_1 \circ \boldsymbol{g}_1 = \boldsymbol{f}_2 \circ \boldsymbol{g}_2$. Consider the coproduct copairing $[\boldsymbol{g}_1, \boldsymbol{g}_2] : \boldsymbol{B}_1+\boldsymbol{B}_2 \rightleftarrows \boldsymbol{C}$. It is straightforward to check that the infomorphism $[\boldsymbol{g}_1, \boldsymbol{g}_2]$ respects the coinvariant J. So there is a unique infomorphism $\boldsymbol{g} : \boldsymbol{P} \rightleftarrows \boldsymbol{C}$ satisfying $\boldsymbol{\tau}_J \circ \boldsymbol{g} = [\boldsymbol{g}_1, \boldsymbol{g}_2]$. Hence, $\boldsymbol{g} : \boldsymbol{P} \rightleftarrows \boldsymbol{C}$ is a unique infomorphism satisfying $\boldsymbol{m}_1 \circ \boldsymbol{g} = \boldsymbol{g}_1$ and $\boldsymbol{m}_2 \circ \boldsymbol{g} = \boldsymbol{g}_2$.
3. In summary, pushouts can be constructed from coproducts and coinvariants.

   coproducts & coinvariants ⇒ pushouts.

   In the other direction, given two classifications $\boldsymbol{A}$ and $\boldsymbol{B}$, the pushout of the initial infomorphisms $!_{\boldsymbol{A}} : \boldsymbol{0} \rightleftarrows \boldsymbol{A}$ and $!_{\boldsymbol{B}} : \boldsymbol{0} \rightleftarrows \boldsymbol{B}$ is the coproduct. So, coproducts can be constructed from pushouts and the initial classification.

   pushouts & initial objects ⇒ coproducts (and more strongly, finite cocompleteness).

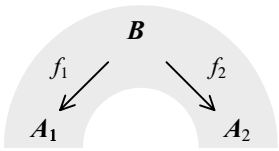


Figure 12: Pushout Diagram = Span

- A *pushout* is a finite limit for a diagram of shape $\bullet \leftarrow \bullet \rightarrow \bullet$. Such a diagram (of classifications and infomorphisms) is called an *span* (see Figure 12, where arrows denote infomorphisms) A *span* is the appropriate base diagram for a pushout. Each opspan consists of a pair of infomorphisms called *first* and *second*. These are required to have a common source classification $\boldsymbol{B}$, denoted as the *vertex*. Let either '`diagram`' or '`span`' be the term that denotes the *Span* collection. Spans are determined by their pair of component infomorphisms.

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

(2) (CNG$function classification1)
    (CNG$signature classification1 diagram CLS$classification)

(3) (CNG$function classification2)
    (CNG$signature classification2 diagram CLS$classification)

(4) (CNG$function vertex)
    (CNG$signature vertex diagram CLS$classification)

(5) (CNG$function first)
    (CNG$signature first diagram CLS.INFO$classification)

(6) (CNG$function second)
    (CNG$signature second diagram CLS.INFO$classification)

    (forall (?r (diagram ?r))
       (and (= (CLS.INFO$source (first ?r)) (vertex ?r))
            (= (CLS.INFO$source (second ?r)) (vertex ?r))
            (= (CLS.INFO$target (first ?r)) (classification1 ?r))
            (= (CLS.INFO$target (second ?r)) (classification2 ?r))))

    (forall (?r1 (diagram ?r1) ?r2 (diagram ?r2))
       (=> (and (= (first ?r1) (first ?r2))
                (= (second ?r1) (second ?r2)))
           (= ?r1 ?r2)))
```

- The *pair* of source classifications (suffixing discrete diagram) underlying any span (pushout diagram) is named.

```
(7) (CNG$function pair)
    (CNG$signature pair diagram CLS.COL.COPRD$diagram)
    (forall (?r (diagram ?r))
        (and (CLS.COL.COPRD$classification1 (pair ?r)) (classification1 ?r))
             (CLS.COL.COPRD$classification2 (pair ?r)) (classification2 ?r))))
```

- Every span has an opposite.

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

    (forall (?r (span ?r))
       (and (= (classification1 (opposite ?r)) (classification2 ?r))
            (= (classification2 (opposite ?r)) (classification1 ?r))
            (= (vertex (opposite ?r)) (vertex ?r))
            (= (first (opposite ?r)) (second ?r))
            (= (second (opposite ?r)) (first ?r))))
```

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

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

- There is an *instance opspan* or *instance diagram* function, which maps a span of infomorphisms to the underlying (`SET.LIM.PBK`) opspan of instance functions. Similarly, there is a *type span* or *type diagram* function, which maps a span of infomorphisms to the underlying (`SET.COLIM.COEQ`) span of type functions.

```
(9) (CNG$function instance-diagram)
    (CNG$function instance-opspan)
    (= instance-opspan instance-diagram)
    (CNG$signature instance-diagram diagram SET.LIM.PBK$diagram)
    (forall (?r (diagram ?r))
        (and (= (SET.LIM.PBK$class1 (instance-diagram ?r))
                (CLS$instance (classification1 ?r)))
             (= (SET.LIM.PBK$class2 (instance-diagram ?r))
                (CLS$instance (classification2 ?r)))
             (= (SET.LIM.PBK$opvertex (instance-diagram ?r))
                (CLS$instance (vertex ?r)))
             (= (SET.LIM.PBK$opfirst (instance-diagram ?r))
                (CLS.INFO$instance (first ?r)))
             (= (SET.LIM.PBK$opsecond (instance-diagram ?r))
                (CLS.INFO$instance (second ?r)))))

(10) (CNG$function type-diagram)
     (CNG$function type-span)
     (= type-span type-diagram)
     (CNG$signature type-diagram diagram SET.COL.PSH$diagram)
     (forall (?r (diagram ?r))
         (and (= (SET.COL.PSH$class1 (type-diagram ?r))
                 (CLS$type (classification1 ?r)))
              (= (SET.COL.PSH$class2 (type-diagram ?r))
                 (CLS$type (classification2 ?r)))
              (= (SET.COL.PSH$vertex (type-diagram ?r))
                 (CLS$type (vertex ?r)))
              (= (SET.COL.PSH$first (type-diagram ?r))
                 (CLS.INFO$type (first ?r)))
              (= (SET.COL.PSH$second (type-diagram ?r))
                 (CLS.INFO$type (second ?r)))))
```

- There is a *parallel pair* or *coequalizer diagram* function, which maps a span of infomorphisms to the associated (`CLS.COL.COEQ`) parallel pair of infomorphisms (see Figure 13, where arrows denote infomorphisms), which are the composite of the first and second infomorphisms of the span with the coproduct injection infomorphisms of the binary coproduct of the component

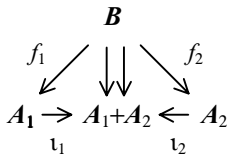


Figure 13: Coequalizer Diagram

classifications. The coequalizer and canon of the associated parallel pair will be used to define the pushout. This parallel-pair is represented by axiom (10).

```
(10) (CNG$function coequalizer-diagram)
     (CNG$function parallel-pair)
     (= parallel-pair coequalizer-diagram)
     (CNG$signature coequalizer-diagram diagram CLS.COL.COEQ$diagram)
     (forall (?r (diagram ?r))
         (and (= (CLS.COL.COEQ$source (coequalizer-diagram ?r))
                 (vertex ?r))
              (= (CLS.COL.COEQ$source (coequalizer-diagram ?r))
                 (CLS.COL.COPRD$binary-coproduct (pair ?r)))
              (= (CLS.COL.COEQ$infomorphism1 (coequalizer-diagram ?r))
                 (CLS.INFO$composition
                     (first ?r)
                     (CLS.COL.COPRD$injection1 (pair ?r))))
              (= (CLS.COL.COEQ$infomorphism2 (coequalizer-diagram ?r))
                 (CLS.INFO$composition
                     (second ?r)
                     (CLS.COL.COPRD$injection2 (pair ?r))))))
```

- There are two important categorical identities (not just isomorphisms) that relate (1) the underlying instance limit and (2) the underlying type colimit of the coequalizer of the parallel pair of any span to (1′) the limit (equalizer) of the equalizer diagram of the underlying instance opspan and (2′) the colimit (coequalizer) of the coequalizer diagram of the underlying type span. These identities are assumed in the definition of the colimiting cocone below.

```
(forall (?r (diagram ?r))
    (and (= (CLS.COL.COEQ$instance-diagram
                (CLS.COL.COEQ$colimiting-cocone (coequalizer-diagram ?r)))
            (SET.LIM.EQU$limiting-cone
                (SET.LIM.EQU$equalizer-diagram (instance-diagram ?r))))
         (= (CLS.COL.COEQ$type-diagram
                (CLS.COL.COEQ$colimiting-cocone (coequalizer-diagram ?r)))
            (SET.LIM.COEQ$colimiting-cocone
                (SET.COL.COEQ$coequalizer-diagram (type-diagram ?r))))))
```

- *Pushout cocones* are used to specify and axiomatize pushouts. Each pushout cocone (Figure 14, where arrows denote infomorphisms) has an underlying *diagram* (the shaded part of Figure 14), an *opvertex* classification $\hat{A}$, and a pair of infomorphisms called *opfirst* and *opsecond*, whose common target classification is the opvertex and whose source classifications are the target classifications of the infomorphisms in the span. The opfirst and opsecond infomorphisms form a commutative diagram with the span. A pushout cocone is the very special case of a colimiting cocone under a span. The term '`cocone`' in axiom (9) is the term that denotes the *Pushout Cocone* collection. The term '`cocone-diagram`' axiomatized in axiom (10) represents the underlying diagram.

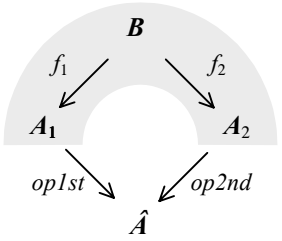


Figure 14: Pushout Cocone

```
(9) (CNG$conglomerate cocone)

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

(11) (CNG$function opvertex)
     (CNG$signature opvertex cocone CLS$classification)

(12) (CNG$function opfirst)
     (CNG$signature opfirst cocone CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
        (and (= (CLS.INFO$source (opfirst ?s)) (classification1 (cocone-diagram ?s)))
             (= (CLS.INFO$target (opfirst ?s)) (opvertex ?s))))

(13) (CNG$function opsecond)
     (CNG$signature opsecond cocone CLS.INFO$infomorphism)
     (forall (?s (cocone ?s))
```

```
      (and (= (CLS.INFO$source (opsecond ?s)) (classification2 (cocone-diagram ?s)))
           (= (CLS.INFO$target (opsecond ?s)) (opvertex ?s))))

   (forall (?s (cocone ?s))
      (= (CLS.INFO$composition (first (cocone-diagram ?s)) (opfirst ?s))
         (CLS.INFO$composition (second (cocone-diagram ?s)) (opsecond ?s))))
```

- The *binary-coproduct cocone* underlying any cocone (pushout diagram) is named.

```
(14) (CNG$function binary-coproduct-cocone)
     (CNG$signature binary-coproduct-cocone diagram CLS.COL.COPRD$cocone)
     (forall (?s (cocone ?s))
        (and (= (CLS.COL.COPRD$opvertex (binary-coproduct-cocone ?r)) (opvertex ?s))
             (= (CLS.COL.COPRD$opfirst (binary-coproduct-cocone ?r)) (opfirst ?s))
             (= (CLS.COL.COPRD$opsecond (binary-coproduct-cocone ?r)) (opsecond ?s))))
```

- There is a *coequalizer cocone* function, which maps a cocone of infomorphisms to the associated (`CLS.COL.COEQ`) coequalizer cocone of infomorphisms (see Figure 15, where arrows denote infomorphisms), which is the binary coproduct comediator of the opfirst and opsecond infomorphisms with respect to the coequalizer diagram of a cocone. This is the first step in the definition of the pushout comediator infomorphism. This coequalizer cocone is represented by axiom (14). The following string of equalities demonstrates that this cocone is well-defined.

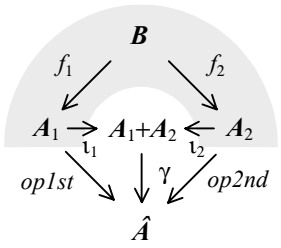


Figure 15: Coequalizer Cocone

$$f_1 \cdot \iota_1 \cdot \gamma = f_1 \cdot op1st = f_2 \cdot op1st = f_2 \cdot \iota_2 \cdot \gamma$$

```
(14) (CNG$function coequalizer-cocone)
     (CNG$signature coequalizer-cocone diagram CLS.COL.COEQ$cocone)
     (forall (?r (diagram ?r))
         (and (= (CLS.COL.COEQ$cocone-diagram (coequalizer-cocone ?r))
                 (coequalizer-diagram ?r))
              (= (CLS.COL.COEQ$opvertex (coequalizer-cocone ?r))
                 (opvertex ?r))
              (= (CLS.COL.COEQ$infomorphism (coequalizer-cocone ?r))
                 (CLS.COL.COPRD$comediator (binary-coproduct-cocone ?r))))))
```

- There is a unary CNG function '`colimiting-cocone`' that maps a span to its pushout (colimiting pushout cocone) (see Figure 16, where arrows denote infomorphisms). Axiom (*) gives the definition of this function in terms of the associated coequalizer diagram. This axiom ensures that this pushout is specific. For convenience of reference, we define three terms that represent the components of this pushout cocone. The opvertex of the pushout cocone is a specific *Pushout* classification given by the CNG infomorphism '`pushout`'. It comes equipped with two CNG projection infomorphisms '`injection1`' and '`injection2`'.

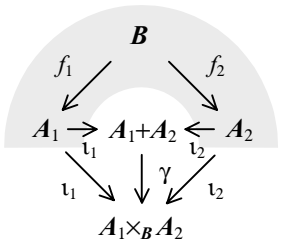


Figure 16: Colimiting Cocone

```
(15) (CNG$function colimiting-cocone)
     (CNG$signature colimiting-cocone diagram cocone)

 (*) (forall (?r (diagram ?r))
         (and (= (cocone-diagram (colimiting-cocone ?r)) ?r))
              (= (opvertex (colimiting-cocone ?r)) ?r))
                 (CLS.COL.COEQ$coequalizer (coequalizer-diagram ?r)))
              (= (opfirst (colimiting-cocone ?r)) ?r))
                 (CLS.INFO$composition
                     (CLS.COL.COPRD$injection1 (pair ?r))
                     (CLS.COL.COEQ$canon (coequalizer-diagram ?r))))
              (= (opsecond (colimiting-cocone ?r)) ?r))
                 (CLS.INFO$composition
                     (CLS.COL.COPRD$injection2 (pair ?r))
                     (CLS.COL.COEQ$canon (coequalizer-diagram ?r))))))

(16) (CNG$function colimit)
```

```
        (CNG$function pushout)
        (= pushout colimit)
        (CNG$signature colimit diagram CLS$classification)
        (forall (?r (diagram ?r))
           (= (colimit ?r) (opvertex (colimiting-cocone ?r))))

(17)    (CNG$function injection1)
        (CNG$signature injection1 diagram CLS.INFO$infomorphism)
        (forall (?r (diagram ?r))
           (and (= (CLS.INFO$source (injection1 ?r)) (classification1 ?r))
                (= (CLS.INFO$target (injection1 ?r)) (colimit ?r))
                (= (injection1 ?r) (opfirst (colimiting-cocone ?r)))))

(18)    (CNG$function injection2)
        (CNG$signature injection2 diagram CLS.INFO$infomorphism)
        (forall (?r (diagram ?r))
           (and (= (CLS.INFO$source (injection2 ?r)) (classification2 ?r))
                (= (CLS.INFO$target (injection2 ?r)) (colimit ?r))
                (= (injection2 ?r) (opsecond (colimiting-cocone ?r)))))
```

- There is a *comediator* infomorphism from the pushout of a span to the opvertex of a cocone over the span (see Figure 17, where arrows denote infomorphisms). This is the unique infomorphism that commutes with opfirst and opsecond.

```
(19)    (CNG$function comediator)
        (CNG$signature comediator cocone CLS.INFO$infomorphism)
        (forall (?s (cocone ?s))
            (and (= (CLS.INFO$source (comediator ?s))
                    (colimit (cocone-diagram ?s)))
                 (= (CLS.INFO$target (comediator ?s))
                    (opvertex ?s))
                 (= (comediator ?s)
                    (CLS.COL.COEQ$comediator
                        (coequalizer-cocone ?s)))))
```

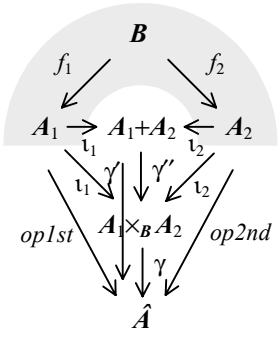


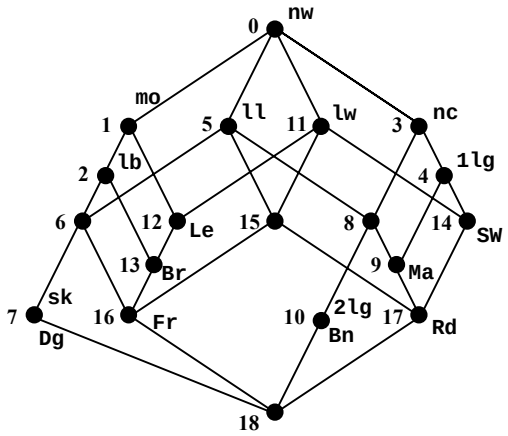
Figure 17: Comediator

Examples

The Living Classification

- The *Living Classification* is a tiny dataset, which exists within a conceptual universe of living organisms. This classification listed below consists of eight organisms (plants and animals), and nine of their properties. The organisms are the *instances* of the classification, and the properties are the *types*. The classification relation is presented as a Boolean matrix in the following table. The living concept lattice, which contains 19 formal concepts, is presented in the following image.

Concept Lattice



Type Class

0	nw	needs water
1	lw	lives in water
2	ll	lives on land
3	nc	needs chlorophyll
4	2lg	2 leaf germination
5	1lg	1 leaf germination
6	mo	is motile
7	lb	has limbs
8	sk	suckles young

Instance Class

0	Le	Leech
1	Br	Bream
2	Fr	Frog
3	Dg	Dog
4	SW	Spike Weed
5	Rd	Reed
6	Bn	Bean
7	Ma	Maize

Classification Relation

	nw	lw	ll	nc	2lg	1lg	mo	lb	sk
Le	×	×					×		
Br	×	×					×	×	
Fr	×	×	×				×	×	
Dg	×		×				×	×	×
SW	×	×		×		×			
Rd	×	×	×	×		×			
Bn	×		×	×	×				
Ma	×		×	×		×			

The following table lists the formal concepts of the Living concept lattice in terms of their extent, intent, instance generators and type generators.

Formal Concepts				
	Generators			
Index	Objects	Attributes	Extent	Intent
0		needs water	{Leech, Bream, Frog, Dog, Spike Weed, Reed, Bean, Maize}	{needs water }
1		is motile	{Leech, Bream, Frog, Dog}	{needs water, is motile }
2		has limbs	{Bream, Frog, Dog}	{needs water, is motile, has limbs }

3		needs chlorophyll	{Spike Weed, Reed, Bean, Maize}	{needs water, needs chlorophyll }
4		1 leaf germination	{Spike Weed, Reed, Maize}	{needs water, needs chlorophyll, 1 leaf germination }
5		lives on land	{Frog, Dog, Reed, Bean, Maize}	{needs water, lives on land }
6			{Frog, Dog }	{needs water, lives on land, is motile, has limbs }
7	Dog	suckles young	{Dog }	{needs water, lives on land, is motile, has limbs, suckles young }
8			{Reed, Bean, Maize }	{needs water, lives on land, needs chlorophyll }
9	Maize		{Reed, Maize }	{needs water, lives on land, needs chlorophyll, 1 leaf germination }
10	Bean	2 leaf germination	{Bean }	{needs water, lives on land, needs chlorophyll, 2 leaf germination }
11		lives in water	{Leech, Bream, Frog, Spike Weed, Reed }	{needs water, lives in water }
12	Leech		{Leech, Bream, Frog }	{needs water, lives in water, is motile }
13	Bream		{Bream, Frog }	{needs water, lives in water, is motile, has limbs }
14	Spike Weed		{Spike Weed, Reed }	{needs water, lives in water, needs chlorophyll, 1 leaf germination }
15			{Frog, Reed }	{needs water, lives in water, lives on land }
16	Frog		{Frog}	{needs water, lives in water, lives on land, is motile, has limbs }
17	Reed		{Reed }	{needs water, lives in water, lives on land, needs chlorophyll, 1 leaf germination }
18			{} = ∅	{needs water, lives in water, lives on land, needs chlorophyll, 2 leaf germination, 1 leaf germination, is motile, has limbs, suckles young }

- The following KIF represents the *Living Classification*.

```
(CLS$Classification Living)
((CLS$type Living) needs-water)
((CLS$type Living) lives-in-water)
((CLS$type Living) lives-on-land)
((CLS$type Living) needs-chlorophyll)
((CLS$type Living) 2-leaf-germination)
((CLS$type Living) 1-leaf-germination)
((CLS$type Living) is-motile)
((CLS$type Living) has-limbs)
((CLS$type Living) suckles-young)
((CLS$instance Living) Leech)
((CLS$instance Living) Bream)
((CLS$instance Living) Frog)
((CLS$instance Living) Dog)
((CLS$instance Living) Spike-Weed)
((CLS$instance Living) Reed)
```

```
((CLS$instance Living) Bean)
((CLS$instance Living) Maize)
...
((CLS$incidence Living) [Leech needs-water])
((CLS$incidence Living) [Leech lives-in-water])
(not ((CLS$incidence Living) [Leech lives-on-land]))
(not ((CLS$incidence Living) [Leech needs-chlorophyll]))
(not ((CLS$incidence Living) [Leech 2-leaf-germination]))
(not ((CLS$incidence Living) [Leech 1-leaf-germination]))
((CLS$incidence Living) [Leech is-motile])
(not ((CLS$incidence Living) [Leech has-limbs]))
(not ((CLS$incidence Living) [Leech suckles-young]))
...
```

- The *Living Lattice* is the concept lattice of the Living Classification.

```
(CL$concept-lattice Living-Lattice)
(= Living-Lattice (CLS.CL$concept-lattice Living))
```

The Dictionary Classification

Here are examples of classifications and infomorphisms taken from the text *Information Flow: The Logic of Distributed Systems* by Barwise and Seligman.

- The following KIF represents the *Webster Classification* on page 70 of Barwise and Seligman. This classification, which is (a small part of) the classification of English words according to parts of speech as given in Webster's dictionary, is diagrammed on the right.

Table 4: Webster Classification

Webster	Noun	Int-Vb	Tr-Vb	Adj
bet	1	1	1	0
eat	0	1	1	0
fit	1	1	1	1
friend	1	0	1	0
square	1	0	1	1
…			…	

```
(CLS$Classification Webster)
((CLS$type Webster) Noun)
((CLS$type Webster) Intransitive-Verb)
((CLS$type Webster) Transitive-Verb)
((CLS$type Webster) Adjective)
((CLS$instance Webster) bet)
((CLS$instance Webster) eat)
((CLS$instance Webster) fit)
((CLS$instance Webster) friend)
((CLS$instance Webster) square)
...
((CLS$incidence Webster) [bet Noun])
((CLS$incidence Webster) [bet Intransitive-Verb])
((CLS$incidence Webster) [bet Transitive-Verb])
(not ((CLS$incidence Webster) [bet Adjective]))
(not ((CLS$incidence Webster) [eat Noun]))
((CLS$incidence Webster) [eat Intransitive-Verb])
((CLS$incidence Webster) [eat Transitive-Verb])
(not ((CLS$incidence Webster) [fit Adjective]))
((CLS$incidence Webster) [fit Noun])
((CLS$incidence Webster) [fit Intransitive-Verb])
((CLS$incidence Webster) [fit Transitive-Verb])
((CLS$incidence Webster) [fit Adjective])
((CLS$incidence Webster) [friend Noun])
(not ((CLS$incidence Webster) [friend Intransitive-Verb]))
((CLS$incidence Webster) [friend Transitive-Verb])
(not ((CLS$incidence Webster) [friend Adjective]))
((CLS$incidence Webster) [square Noun])
(not ((CLS$incidence Webster) [square Intransitive-Verb]))
((CLS$incidence Webster) [square Transitive-Verb])
((CLS$incidence Webster) [square Adjective])
...
```

- The following KIF represents the infomorphism defined on page 73 of Barwise and Seligman. This represents the way that punctuation at the end of a sentence carries information about the type of the sentence. The infomorphism is from a *Punctuation* classification to a *Sentence* classification. The instances of *Punctuation* are the inscriptions of the punctuation marks of English. These marks are

classified by the terms ‘Period, ‘Exclamation-Mark’, ‘Question-Mark’, ‘Comma’, etc. The instances of *Sentence* are inscriptions of grammatical sentences of English. There are but three types of *Sentence*: ‘Declarative’, ‘Question’, and ‘Other’. The instance function of the infomorphism assigns to each sentence its own terminating punctuation mark. The type function of the infomorphism assigns ‘Declarative’ to ‘Period’ and ‘Exclamation-Mark’, ‘Question’ to ‘Question-Mark’, and ‘Other’ to other types of *Punctuation*. The fundamental property of this infomorphism is the requirement that a sentence be of the type indicated by its punctuation.

Let ‘yakity-yak’ denote the command “Take out the papers and the trash!” with its punctuation symbol ‘yy-punc’ being the exclamation symbol at the end of the sentence. Let ‘gettysburg1’ denote the statement that “Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty and dedicated to the proposition that all men are created equal.” with its punctuation symbol ‘g1-punc’ being the period at the end of the sentence. Let ‘angels’ denote the question “How many angels can fit on the head of a pin?” with its punctuation symbol ‘ag-punc’ being the question mark at the end of the sentence.

```
(CLS$Classification Punctuation)
((CLS$type Punctuation) Period)
((CLS$type Punctuation) Exclamation-Mark)
((CLS$type Punctuation) Question-Mark)
((CLS$type Punctuation) Comma)
((CLS$instance Punctuation) yy-punc)
((CLS$instance Punctuation) g1-punc)
((CLS$instance Punctuation) ag-punc)
...
(not ((CLS$incidence Punctuation) [yy-punc Period]))
((CLS$incidence Punctuation) [yy-punc Exclamation-Mark])
(not ((CLS$incidence Punctuation) [yy-punc Question-Mark]))
...

(CLS$Classification Sentence)
((CLS$type Sentence) Declarative)
((CLS$type Sentence) Question)
((CLS$type Sentence) Other)
((CLS$instance Sentence) yakity-yak)
((CLS$instance Sentence) gettysburg1)
((CLS$instance Sentence) angels)
...
((CLS$incidence Sentence) [yakity-yak Declarative])
(not ((CLS$incidence Sentence) [yakity-yak Question]))
(not ((CLS$incidence Sentence) [yakity-yak Other]))
...
(CLS.INFO$Infomorphism punct-type)
(= (CLS.INFO$source punct-type) Punctuation)
(= (CLS.INFO$target punct-type) Sentence)

(= ((CLS.INFO$instance punct-type) yakity-yak) yy-punc)
(= ((CLS.INFO$instance punct-type) gettysburg1) g1-punc)
...
(= ((CLS.INFO$type punct-type) Period) Declarative)
(= ((CLS.INFO$type punct-type) Exclamation-Mark) Declarative)
(= ((CLS.INFO$type punct-type) Question-Mark) Question)
(= ((CLS.INFO$type punct-type) Comma) Other)
...
```

The Truth Classification

- The *truth classification* of a first-order language ***L*** is the large classification, whose instances are ***L***-structures (models), whose types are ***L***-sentences, and whose classification relation is satisfaction. Here we represent the truth classification in an external namespace. Note that the source is a class, whereas the target is a conglomerate – rather unusual. The image should then be just a class.

```
(CNG$function truth-classification)
(CNG$signature truth-classification lang$language CLS$classification)
(forall (?l (lang$language ?l))
    (and (= (CLS$instance (truth-classification ?l))  (MOD$fiber ?l))
         (= (CLS$type (truth-classification ?l))      (lang$sentence ?l))
         (= (CLS$incidence (truth-classification ?l)) (MOD$satisfaction ?l))))
```

- The *truth lattice* is the concept lattice of the truth meta-classification. This complete lattice functions as an 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 join of two theories (concepts) is the intersection of the theories (intents), and the meet of two theories is the theory of the common models.

```
(CNG$function truth-lattice)
(CNG$signature truth-lattice lang$language CL$concept-lattice)
(forall (?l (lang$language ?l))
    (= (truth-lattice ?l)
       (CLS.CL$concept-lattice (truth-classification ?l))))
```

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

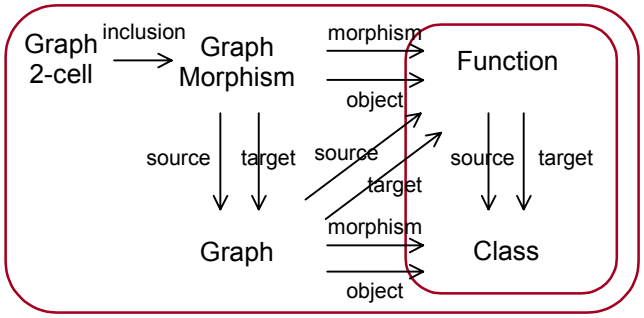
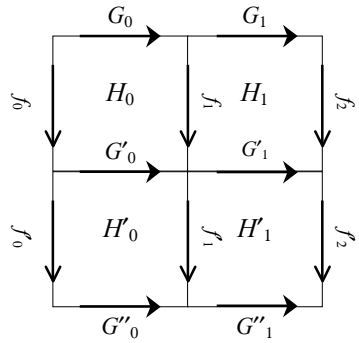


Diagram 1: Core Conglomerates and Functions

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 $src(G_1) : mor(G_1) \to obj$.

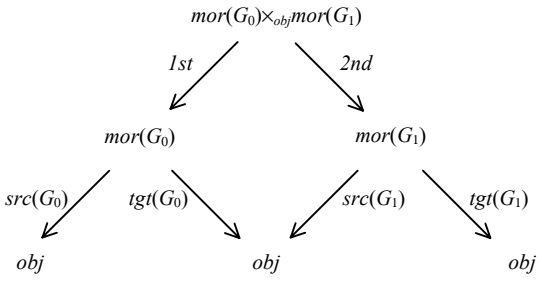


Figure 2: Multiplication Graph

$mor(G_0) \times_{Obj} mor(G_1) =$
$\{(m_0, m_1) \mid m_0 \in mor(G_0)$ and $m_1 \in mor(G_1)$ 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.

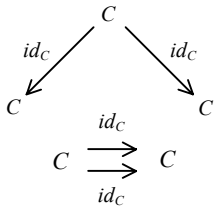


Figure 3: Unit Graph

```
(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 $obj(H) : Obj(G) \to Obj(G')$ assigns to each object of G an object of G', and the morphism function $mor(H) : Mor(G) \to Mor(G')$ assigns to each morphism of G

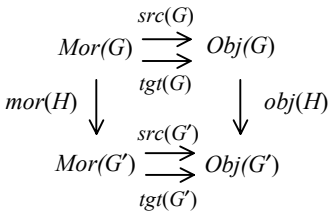


Figure 4: Graph Morphism

a morphism 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).

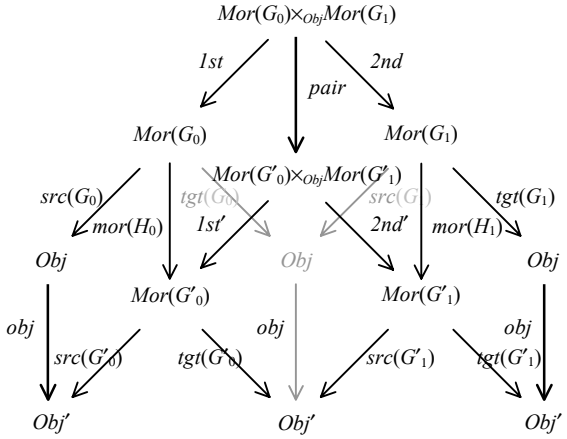


Figure 5: Multiplication Graph Morphism

```
(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.

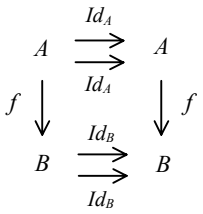


Figure 6: Unit Graph Morphism

```
(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.

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

  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.

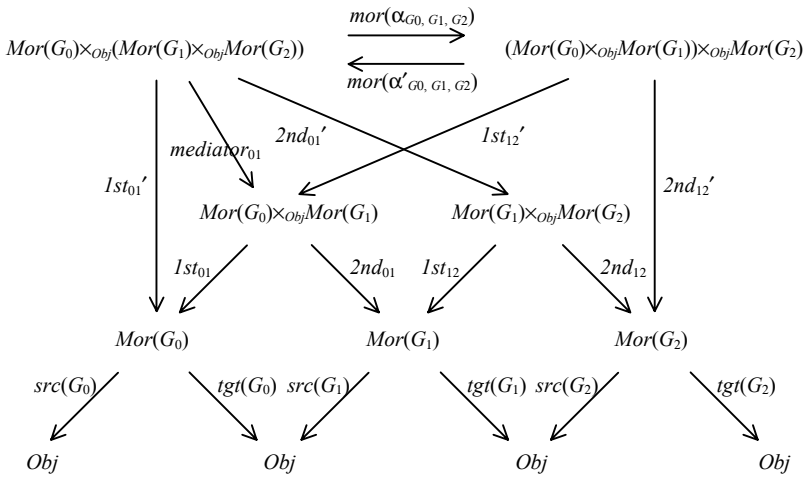


Figure 7: Associativity Graph Isomorphism

In order to define the morphism function

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

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

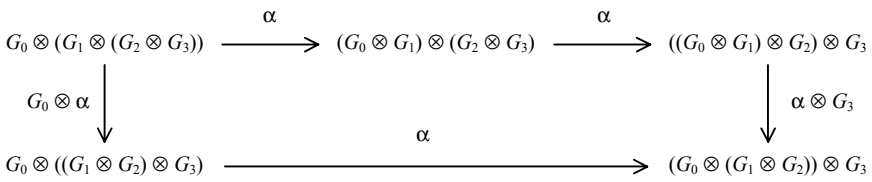


Diagram 3: Associativity Coherence

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.

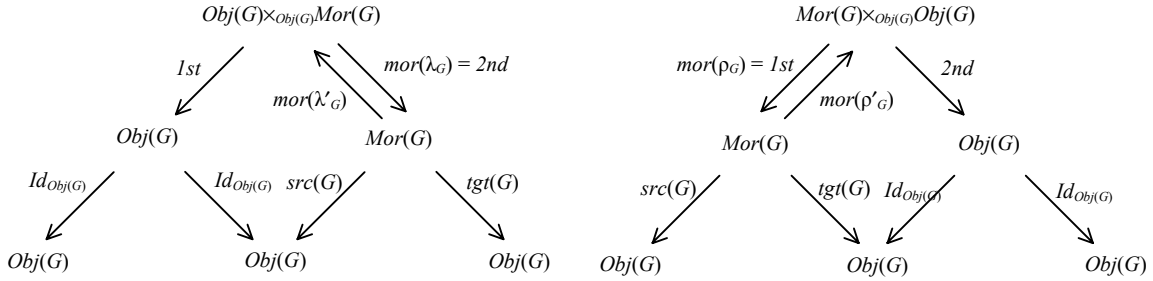


Figure 8: Left Unit and Right Unit Graph Isomorphisms

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

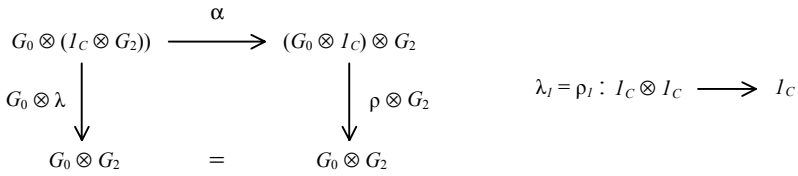


Diagram 4: Unit Coherence

be proven.

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 $\eta_C : 1_{obj(C)} \to |C|$, 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.

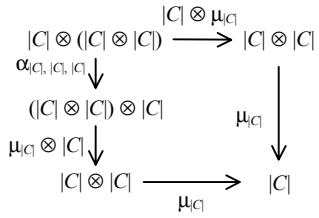
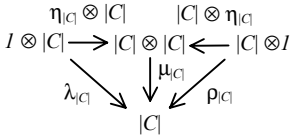
	
Associative Law	**Left/Right Unit Laws**

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)
```

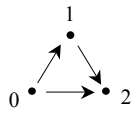


Figure 5: ordinal 3

- 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).

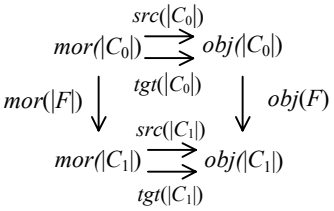


Figure 1: Functor

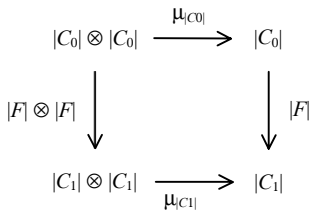
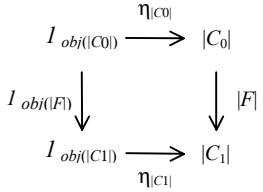
  Axioms (1–4) give the KIF representation for a functor. The unary CNG function '`underlying`' in axiom (4) gives the *underlying* graph morphism of a functor. Functors are determined by their underlying graph morphism.

```
(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.

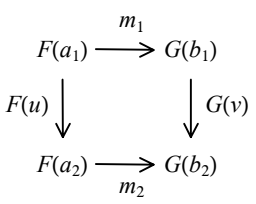


Diagram 1: comma category

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 $(a \downarrow U)$.

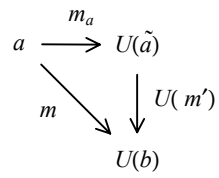


Diagram 2: universal morphism

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* $F \circ G : C_0 \to C_2$. 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 $\Delta_{J,C}(o) = !_J \circ elmt_C(o)$ – 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

  $Id_{C0} \circ F = F$ and $F = F \circ Id_{C1}$

  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.

- 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* $inst : \mathsf{Classification} \to \mathsf{Set}^{op}$ (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* $typ : \mathsf{Classification} \to \mathsf{Set}$ – 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* $pow : \mathsf{Set}^{op} \to \mathsf{Classification}$. 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:

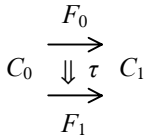


Figure 1: Natural Transformation

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

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).

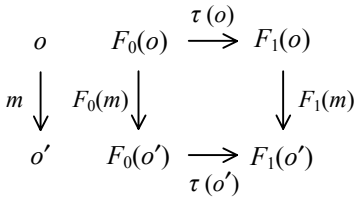


Diagram 1: Natural Transformation

```
(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.

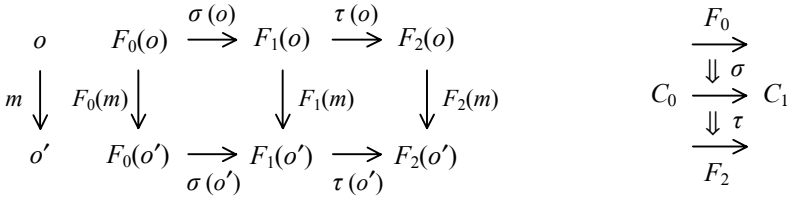


Diagram 2: Vertical composition

- 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.

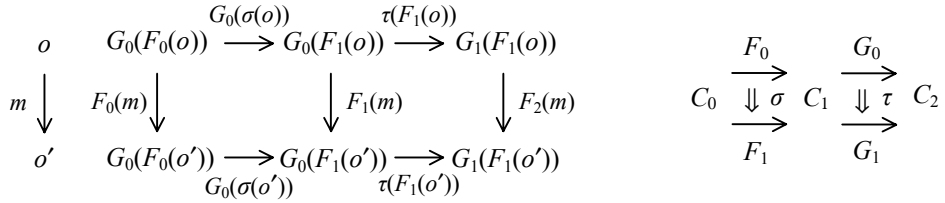


Diagram 3: Horizontal composition

- 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.

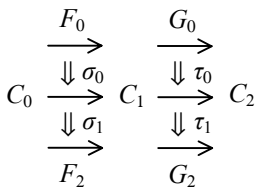


Figure 2: Interchange Law

```
(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.

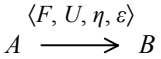


Figure 1: Adjunction

- 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.

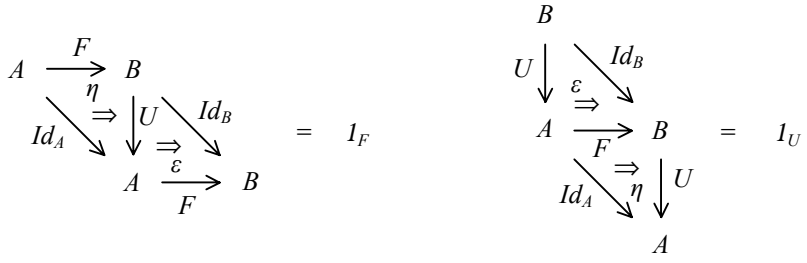


Diagram 1: Triangle identities

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 $F(m : a \to a') = \eta_{\hat{a}} \cdot_A U(F(m)) = m \cdot \eta_{\hat{a}}$.

  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 $\langle F, U, \eta, 1_{IdB} \rangle : A \to B$ 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 $\langle F, U, 1_{IdA}, \varepsilon \rangle : A \to B$ 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:

$$\begin{array}{c} \langle F, U, \eta, \varepsilon \rangle \\ \longrightarrow \\ A \;\Downarrow \langle \sigma, \tau \rangle\; B \\ \longrightarrow \\ \langle F', U', \eta', \varepsilon' \rangle \end{array}$$

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 → Set^{op}

  is left adjoint *inst* ⊣ *pow* to the instance power functor

  pow : Set^{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 $T^3 = T \circ T \circ T$):

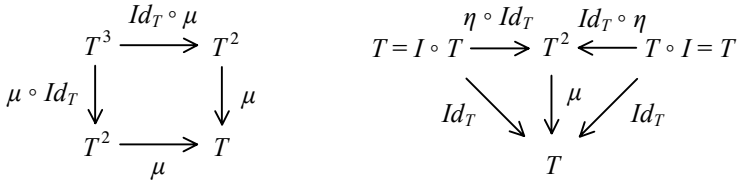


Table 1: Monad

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*.

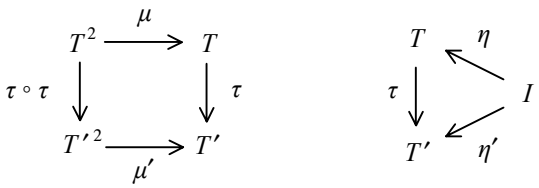


Table 2: Monad Morphism

```
(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*.

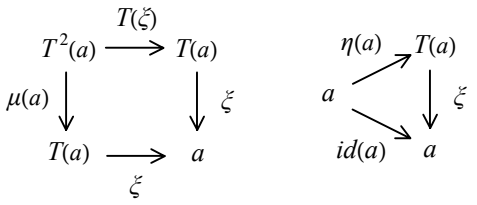


Table 3: Algebra

```
(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. We use a KIF definite description to define the counique function.

$$\underbrace{\quad}_{J} \quad \overbrace{0}^{D}$$

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*.

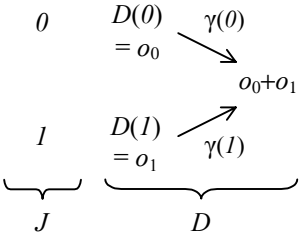


Figure 2: Binary coproduct

- 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

. . .

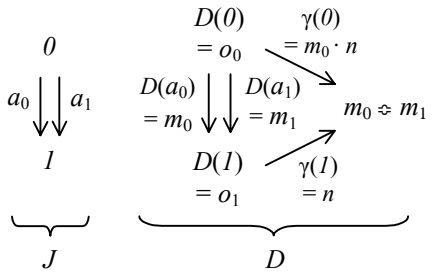


Figure 3: Coequalizer

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.

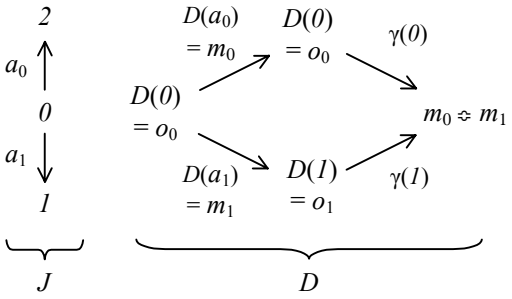


Figure 4: Pushout

```
(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.

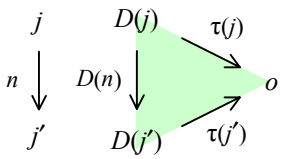


Figure 5: Cocone

```
(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 `opvertex` $\bar{o} \in obj(C)$

  $\gamma : D \Rightarrow \Delta_{C,J}(\bar{o}) : J \to C$

  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.

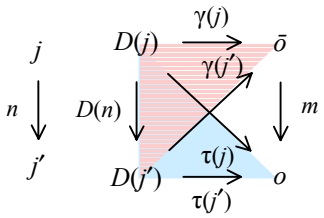


Figure 6: Colimiting Cocone

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 Tracts in Theoretical Computer Science 44. Cambridge University Press.

Davey, B.A. and Priestly, H.A. 1990. ***Introduction to Lattices and Order***. Cambridge Mathematical Textbooks. 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).